SADE 1.3 Release Notes

XCMDs etc.

- SADE's default behavior does not allow you to set breakpoints in resources which are not of the type 'CODE'. You can change this behavior by setting the built-in SADE variable `BreakInNonCODERsrc` to 1. Be warned, however, that as SADE must then check every resource which is being loaded into memory, this option will slow things down.
- You can debug Hypercard XCMDs and XFCNs with SADE. To be able to debug XCMDs, you must remove the resource from the original stack and paste it into the Hypercard application itself. You must also set the SADE variable `BreakInNonCODERsrc` to 1.
- Some applications may load your XCMDs in a subheap. To be able to debug, you need to set another built-in SADE variable, `CurTargetZone`, to the address of that subheap.

Known Bugs

- SADE 1.3 does not support scripts that contain two or more execution commands. However, it also does not prevent you from using such a script. If you execute a script containing multiple execution commands, SADE does not halt execution and return an error message, but rather executes the script with unpredictable results.
- The SADEKey may not work until the target application has made an event call (for instance, WaitNextEvent).

  Pressing the SADEKey multiple times in a row may cause a crash.

  Targeting a running application and then hitting the SADEKey may not work.
- The `Stack` command does not support the documented option `at` address.
- If the `Stack` command is part of a break action, the stack will not necessarily be displayed when the breakpoint is encountered. A Printf command following the stack command will flush internal buffers, forcing any pending output.
- SADE shows variables containing sets as integers, instead of showing the set elements with their names.
- If you kill the target using the menu command, the watch variables are removed and the conditional break table is cleared. However, if you use the SADE command `Kill`, or quit the target, these clean up actions are not done. If you target again, the old watch variables may not be relevant.
- There is a size limit of 1,006 bytes on the value which can be assigned to a SADE variable. As a consequence of this, you can not use the Show Value menu to display a variable of size greater than 1,006 bytes. You can, however, enter the name of the variable you want to display in a writable window and hit enter to get its value.
- Assigning a large C structure to a SADE variable in a define statement may display garbage as part of the error message. For instance, if the variable *myStruct* is a large structure and you execute the following command in the SADE worksheet, an error message will be displayed which contains garbage characters:

```
define myStruct := b
```

- To reference Pascal variables declared at UNIT scope, when the PC is not in the scope of the unit, use the syntax, *\UNITNAME.variablename*. To reference Pascal variables declared at PROGRAM scope use the syntax, *\variableName*; referencing them as *\PROGRAMNAME.variableName* does not work.

- When running SADE on a Classic and targeting a MacApp program, once you step into a MacApp method, SADE can't retrieve the symbol information for the source.
- If Stop Before Constructor is set and after targeting you step over main() in a C or C++ application, quitting the application will produce an error message of "Data Initialization Failed!".
- Stepping over a CHK or TRAPV instruction which will cause an exception results in a crash.
- If you enter SADE by calling a SysError() call, the condition code registers displayed will be invalid.
- Since SADE allows users to use symbol files which may not have been produced as part of the application build, SADE can not do any checking of the symbol file date relative to the application date. If you see the the error message:

  ```
  Logical end-of-file reached during read operation (OS error -39)
  ```

  when you target a process, it probably means that your application and the symbol file are out of sync.
- The current symbolic information does not include information specific to classes. While SADE supports commands like `Break Class` *`TSomething`*, it requires SADE to scan the entire symbol file to find matching names. As a result, this command can take a long time.
- Similarly, the symbolic information does not include information about static class variables, and SADE does not display these when a class object is displayed. To see a class static variable, you have to fully qualify the name, e.g.

  ```
  TClass:fgStaticField
  ```
- Users of Object Pascal who are not using MacApp should always link with `-opt on`. Failure to do so will cause SADE to crash.
- If you have trace points set within the main event loop of your target program, so that they are continually displayed in the worksheet, the target application's menu may occasionally overlap the SADE menu. This is only a cosmetic problem in the Process Manager (System 7.0) and in Multifinder (System 6.0) and should not affect the functionality of your program.

System 7.0 specific bugs:

- Under System 7.0, if you've targeted an application and stepped through your initialization code, even though your target is suspended, hiding SADE will cause the target to run. This is a bug in the Process Manager.
- If a file is open modifiable from a shared volume by one user, that file cannot be opened (read-only) for targeting by another user.
- Do not launch two debuggers at the same time. If you quit the second one launched, and try any operation in the first one, you will get the cryptic message:

```
#   Application made module calls in improper order  (OS error -603)
```

System 6.0 specific bugs:

- Special care needs to be taken when debugging MacApp programs under system 6.0.x. After killing a program the first time, you may not be able to retarget the application and be forced to reboot. This is because MacApp installs a VBL which is not being removed by MultiFinder after a kill. This does not occur under system 7.0. A work around is to always quit your application normally; do not kill the application from SADE.
- SADE doesn't handle script execution properly when the script is in the same folder as the target. Workaround: fully qualify the path to the script before executing it.
- There is a bug in Multifinder 6.1b9 which prevents you from being able to scroll the Apple Menu when the option key is held down.
- If you use the target command when you already have a suspended target, the old target will not be restarted.
- If you launch a target from SADE, and quit SADE without first killing it, when the target quits, Multifinder will try to send a message to a non-existent process and cause a system crash.
- The SADEKey cannot be changed under System 6.0.
- SADE does not work with Maxima™ or Virtual (both from Connectix Corp.).

Compiler and Linker symbol generation bugs:

- The compilers do not generate type information unless considered necessary. If you have a type declared but do not have any object of that type, possibly used only for casting, no symbolics are generated for the type. Workaround: none. This is a compiler feature to minimize the symbol table size.

- CFront typically generates constructor and destructor code at the beginning and end of syntactic blocks, gives them separate symbolic information records, but maps them to the same location in the source file. One observed behavior is stepping multiple times on the same source statement. Also, in this situation if you do a Break followed by an Unbreak, the location you set the breakpoint and the location you did the unbreak may not be the same, even if the selection looks identical. Workaround: none.

- If the source code contains a series of declarations with assignments, CFront produces incorrect symbol information for these. The observed behavior is that you need to single step as many times as there are declarations, but SADE keeps highlighting the first statement in the series. After sufficient number of steps, SADE will jump over all the declaration code at once. Workaround: none.

- C++ allows variables to be declared anywhere within a block. When a program is stopped in a block before a variable comes into scope, SADE will display an incorrect value for the variable. A better behavior might be to say that the variable is not in scope.

- CFront passes a temporary file `C.pipe.code` to the C compiler. The C compiler incorrectly assigns all global static variables to the unit `C`. For this reason, references to static variables in a file other than the current one does not work as expected. Workaround: instead of `\FileName.StaticName`, try `\C.StaticName`.

- SADE can not print the values of anonymous unions in C++ correctly. If you have:

```
struct Outer {
  int field;
  union {  // anonymous
        int cant_find;
        char another_cant_find;
  };
} Outer_thing;
```

  SADE is unable to display the value of `Outer_thing.cant_find`. Workaround: If you look at the struct that CFront makes, you will see a name like `__O1` for the anonymous union. If you say `Outer_thing.__O1.cant_find` then SADE will print it.

- Under some circumstances, CFront 3.1 and MPW C 3.2 compiler working together produces incorrect source statement information. The symptom is SADE's inability to set breakpoint in some routine. Workaround: none.

- Both the MPW 3.2 C and Pascal compilers may assign multiple user variables to the same register. As a consequence, if you try to display a register variable whose value is currently dead, the value displayed may be wrong.Workaround: none.
- If an enclosed Pascal procedure does not use any variable from the calling procedure, the MPW 3.2 Pascal compiler optimizes away the static link, but does not reflect this in the symbol information, resulting in incorrect display of arguments in the called function. Workaround: use `-opt on,nostatic` or the `-opt off` flags to the Pascal compiler.
- The MPW 3.2 Pascal compiler optimizes away the `LINK` instruction for functions which do not need stack storage for local variables, resulting in incorrect display of the stack. Workaround: give `-opt off` flag to the Pascal compiler.
- The MPW 3.2 Pascal compiler generates incorrect symbol information about the location of local variables that are assigned to a FPU register . Workaround: none.
- The MPW 3.2 Pascal compiler generates incorrect symbol information for enclosed functions with a procedure parameter. Workaround: none.
- If your program contains single-statement `for` loops which should execute multiple times, SADE will appear to step through the loop only once. This is due to the way the both the MPW 3.2 C and Pascal compilers generate symbol information. Workaround: set a breakpoint on the statement in the `for` loop and execute the `Go` command multiple times.
- The MPW C and Pascal compilers occasionally generate incorrect symbol information for statement boundaries. Workaround: give `-opt off` flag to the compiler.
- The current symbolic structure is some what limited in the size of the program that can be compiled with symbolics. The following table lists the number of different symbolic items a symbol file can contain.

Indexes to source statements (CSNTE)	4 G lines	
Number of different source files	2K	
Number of Functions (MTE)	32K	
Number of blocks (CMTE)	32K	The total number of items of
Number of Variables and Constants (CVTE)	32K	these 5 groups can not
Number of Types (TINFO)	32K	exceed 64K
Number of Labels (CLTE)	32K	

- Even if you stay within the above limits, the MPW 3.2 linker occassionally produces incorrect type information in large symbol tables. The `Show Value` menu item(s) normally traps all internal errors and displays the message `Undefined/Out of scope`. However, if you type the name of the object on the worksheet and hit enter, SADE may then print the error message:

  ```
  Symbol table error: Variable type table index could not be found.
  ```

  Workaround: The post 3.2 version of the linker shipped on the ETO#5 disk fixes most of these problems. If you still find this problem, and if your program uses a rich hierarchy of classes, there is no workaround. Even with a partially broken symbol file, you may be able to do most of the things you need to do. If your program is just big, if the size of your symbol table exceeds 1 megabyte, consider compiling parts of application without symbols, and compile only the suspect parts with symbols.

  You can use the Sade diagnostic command `List Symbol` to see the local variables and types that are defined in the current scope. This command will also show if any symbolic information associated with these objects is inconsistent.

- Types which have global scope sometimes get bound to a local scope, you can see the type definition only if the type is used in the current scope, or a containing scope. Workaround: none.

Changes Since SADE 1.3B2

- CFront generates incorrect symbols for any PascalObject with a forward declaration. In the member functions, CFront says that `this` is a `*TType`, then casts every references through `this` to `(**(TType**)this)`. To alleviate this problem some hacks were put in SADE. If your program contains the resource `%__MethTable` (as is the case with MacApp programs), and if you have a type which the symbol file specifies to be a pointer to a PascalObject (a record whose first element is a `__vtable`), SADE will make itself believe that the type really is a pointer to a pointer to that object. As a consequence of this fix, if you singly dereference a handle to a PascalObject, the pointer value displayed will be correct, but the type displayed will be a handle.

  The newly provided menu item `Twice Dereference Selection (⌘-5)`, explained below, makes displaying objects through handles much easier.

- A new menu called `Extras` was added. Four of the menu items under this menu make displaying objects easier, and the last one makes changing the value of objects easier.

  • `Show Value In Context (⌘-2)` and

  • `Show Dereferenced Value In Context (⌘-3)`

  The old style Show Value menu items have no context, they simply follow scoping rules. Thus, if you select `fNext`, it will try to show you an automatic `fNext`, a global `fNext`, `this^.fNext` or `this^^.fNext`.

  These new menu items put the value of the selected variable in a separate window. For subsequent selections, the name of the window provides the context. E.g. if you select `fNext` in your source file and do a ⌘-2, you will most likely get `this^^.fNext`. However, if you had previously displayed a variable `newWindow`, and select `fNext` in that window, ⌘-2 will show you `newWindow.fNext`.

  Since these windows are really files, when the length of the name of the thing to be displayed exceeds 32 characters, it will be displayed in the Value window instead. These menus do not work if your selection has a colon (:) in it. These commands are slightly slower than the regular show value menu items, but having separate window for each variable far outweighs the slight speed penalty.

  • `Set Value (⌘-=)`

  Changing variables has always been difficult in SADE . The problem is that SADE shows you structured values, but you probably need to change only a field in a structured object. It would be nice to type over a displayed value in a window to the

value you want. Unfortunately, it is not that simple.

If you are trying to change a simple variable, simply highlight its name and select the menu item `Set Value`. From the displayed dialog box, you can verify that SADE is going to change the variable you wanted. If the proper object is selected, type in the value you want. You can type in expressions instead of simple numbers. You can also type in the name of a complex object whose type matches that of the left hand side.

In implementing this operation, SADE first evaluates the left hand side, then evaluates the expression you typed in, and may display error messages if it finds any problems. After that SADE will try the actual assignment; that too may fail because of incompatible types for the left and right hand sides.

To change a field buried in a structure, you have to peel the outer levels. E.g. if you want to change

```
FirstLevel.SecondLevel->Member
```

do a ⌘-2 after selecting FirstLevel. This gives you a window named `FirstLevel`. In that window, select the word `SecondLevel` and do a ⌘-3. Now you have a window named `FirstLevel.SecondLevel^`. In this window select the text `Member` and select Set Value or ⌘-=.

Two new menu commands were added to the `Extras` menu to make debugging MacApp applications written in C++ easier.

• The menu item `Show Self (⌘-4)` displays `**this` in a separate window. You must be stopped inside a MacApp method for this menu to work.

C++ programmers who are not using MacApp may wish to edit the SADE routine `__ShowSelf__` in the file SadeStartup file so that this menu displays `*this` instead.

• The menu item `Twice Dereference Selection (⌘-5)` makes displaying objects through a handle to them one step process. The value is displayed in a separate window, as in the previous commands.

This menu is implemented in the file SadeUserStartup•Extras. You can remove this file without affecting SADE behavior in any other way.

n The order in which SADE startup files are executed is:

• SadeStartup

• SadeUserStartup

• all files whose names starts with SADEUserStartup•. These files are executed in alphabetical order.
If you want to override anything defined in the file SadeUserStartup•Extras, change the name of your SadeUserStartup to something like SadeUserStartup•zzz, so that it is alphabetically later.

- SADE will place all temporary files in the sub-folder called `SADE Scratch` inside the SADE launch directory. If the folder does not exist, SADE will create it. During exit, SADE will delete all SADE created files from this directory.

  One side effect of this behavior is that SADE will no longer remember the position of the temporary windows like Values. If you prefer the old behavior, override the scratch directory by adding the following line to your `SADEUserStartup` file:

  ```
  __ScratchDir__ := SADEDir
  ```

- When SADE needs to open a source file, it tries much harder now. The symbol file provides SADE with either a full path name or just the leaf name, depending on the way the file was compiled. In addition, the modification date of the source file when it was last compiled is also provided. Here is the list of actions SADE goes through to find a source file:

  • Check if any of the open files have the same leaf name and matching date. If not,

  • If the symbol table specified a full path name, try the full path to open the file. If not,

  • Try to find a file in the default directory with the same leaf name. If not,

  • Try to find a file in any of the directories in the SourcePath. If not,

  • Check if any of the open files have the same leaf name but a wrong date. If not,

  • Ask the user to navigate the directory structure to find a file with the same leaf name.

  Check the modification date of the file found in any of the above ways, if there is a mismatch, give user a chance to find a better match. If the user finds a file in a new directory, add the new directory to the source path.

- If you open a source file in a directory which is not in the current source path, and try to set a break point in it, if SADE finds the symbol for a file with that name, SADE will add the new directory to the source path.

- When SADE displays a record, the name of the record is displayed in addition to its contents. Occasionally SADE may display names like `__032`, these are generated by CFront in situations like anonymous unions or structs.

- When the conditional break table is full, we shift the oldest break point out instead of deleting all old conditional breakpoints.

Changes Since SADE 1.3B1

- Targeting an application is much simpler now. From the Finder, double-clicking on a SYM file will launch SADE, set the directory to the SYM file's location and target your application. You can also use the 7.0 feature of dragging your application's SYM file or executable file over the SADE application icon to do the same. In addition, you can target an MPW tool by dragging the tool over the SADE application icon. SADE will then prompt you to locate the MPW Shell.

- When using the Target menu item under the File menu, SADE will now display both the application file and the Symbol file. This way, you can target the application by selecting either file. Then, if both files are not in the same directory, SADE will prompt you to locate the missing file. For instance, if you select an application to target and the symbol file is not in the same directory as the application, SADE will put up a dialog asking you to locate the file *AppName.SYM*.

- SADE will now set the Sourcepath variable for you. If SADE cannot find the application's source files in the same directory as the symbol file, it will prompt you to find the source file. In addition, while debugging your code, if you step into a procedure defined in a separate source file and that file is in a different directory than those defined by the sourcepath variable, SADE will prompt you to search for that file as well. SADE will add the new directory indicated to the sourcepath.

  Note, however, that SADE does not clear the Sourcepath variable each time you kill your application. This way, you can continue targeting the same application without SADE prompting you each time to locate your source files.

- SC7 is a new script available in the SADE folder. When executed from the worksheet, you may use the procedure `StackCrawl7` or `sc7` defined in the script to show a possible calling chain sequence. StackCrawl7 is similar to the MacsBug command sc7. It is useful for displaying the calling sequence when the built in A6 based stack command does not work.

- MacsBug and ROM symbols will now be displayed in a disassembly. When there is no symbolic information available and your code has been compiled with MacsBug symbols, SADE will display MacsBug information, e.g.:

```
BEQ.S     µINITIALIZE+$00BA   ; 00BE7B22
```

- SADE will now display the first 20 elements of an array instead of just the first element.
- When displaying the value of a pointer to a string, SADE will always display the pointer value, then the string if it meets the criteria described below.
- C does not have a built-in string type. The following heuristic is employed to determine if a signed or unsigned character array contains a string:

```
// Get past the printable characters
for (i = 1; i < len; i++)
      if (!isgraph(string[i]) && !isspace(string[i]))
            break;

// Is this a CStr?  Must terminate with a null.
if (string[i] == 0)
      // The first byte must be printable
      if (isgraph(string[0]) || isspace(string[0]))
            goto display_CStr;

// Is this a PStr?  The string[length] byte must be printable
if (string[0] < i)
      goto display_PStr;
```

  The significance of this is that if you have partially constructed strings in a char array, SADE may decide you do not have a string. If string display fails, SADE will show you the first 20 elements of your array, which may give you a clue as to why the array did not display as a string. One peculiar case is when the first element of your array contains a zero, in which case SADE will believe that it successfully displayed a null string, and will not show the rest of the array. In most cases this is the right answer, but not always.

  Also note that you can type coerce your array to display it as a PString:

```
^PString(@YourArray)
```

  Similarly, you can coerce your array to display it as a CString:

```
^CString(@YourArray)
```

  Note that Pascal strings are shown with single quotes, e.g. `'string'`, whereas C style null-terminated strings are shown as `"string"`.

- The `Quit` command is now defined in the SADEStartup file. By default, the `Quit` command saves changes to files which are open as read/write before quitting. You can change the script to get the behavior you want.

Other Changes Since SADE 1.2

Warning	There was a major change between versions 1.2 and 1.3 in how SADE handles execution commands in scripts. If you have written SADE 1.2 debugging scripts, be sure to read the new manual.

- The `Target` command now identifies the target program, launches it, stops at the first line of the main program, and opens the source file containing `main`. In version 1.2, `Target` only identified the target program.
- If you quit SADE while your target application is suspended, the target will be killed as well. If the target is currently running, it will continue to execute after SADE has quit.

  If your target is an MPW Tool, the **`Kill`** item under the File menu will now kill a suspended tool, not the MPW Shell. If you are debugging an MPW tool and you want MPW to continue running after quitting SADE, remember to kill the (suspended) tool first, either by using the **`Kill`** menu command or by executing `KillTool` from the worksheet.
- The `Disasm` command now correlates source statement numbers with assembly code instructions. Branch instructions which use the new branch island option, are shown with a ^ character to denote the indirection.
- When you add a watch variable, SADE opens the Variable Watch window and displays the value of the specified variable immediately.
- A new item has been added to the File menu, **`Stop Before Constructor`**. This command is useful for debugging C++ programs which contain static class objects. If this option is selected, SADE will halt before any user code is executed, thus allowing you to set breakpoints, etc. in your constructor code.
- The Stack window is displayed when you select **`Show Stack`** under the Variables menu. The menu now displays a warning if the program counter points to a LINK instruction. When the menu item **`Live Stack Window`** is checked, SADE will update the stack display each time SADE is entered.
- The menu item **`Display Registers`** displays the contents of the data, address, condition code, and program counter registers. The menu item **`Live Register Window`** will update the Register Window each time SADE is entered.
- If you target an alias file of an application file, source file, or SYM file, SADE will use the original file for debugging.

- A new item under the SourceCmds menu, **Break if No Source**, is provided to allow you to determine the behavior of `Step into` when no source statement information is available for the code that was stepped into. This is accomplished with the variable `BreakIfNoSource`. The default value for `BreakIfNoSource`—as set in the SADEStartup file—is 0 (FALSE). When `BreakIfNoSource` is 0, a `Step Into` will continue stepping until there is source statement information available for the instruction at the program counter. The benefit of this is that SADE will not break in out-of-line arithmetic routines such as `ULMODT` and display mysterious lines of assembler. A further benefit is that Object Pascal and MacApp method dispatches are treated as indivisible operations: stepping into a method call will break at the first instruction of the method, without the necessity of using the `StepMethod` procedure.

  The down side (there's always a down side) is that if SADE steps into something without statement information, it will appear that SADE has hung. It hasn't, it's just very busy executing your code 2 to 3 orders of magnitude slower than normal. This down side should only affect you if you compile some of your code with symbols and some without and then inadvertently step into it. Should you do so, you have a choice of waiting for completion of stepping or rebooting. Setting `BreakIfNoSource` to 1 will make SADE break when no source information is available for the PC (this was always the behavior in SADE 1.2).

- The `Go while` and `Go until` commands (that is, the `while` and `until` options of the `Go` command) were removed between versions 1.2 and 1.3. The reason is that they no longer work given the new method by which the SADE interpreter executes commands.

- You no longer need to execute the `StepMethod` procedure (required under SADE version 1.2) to step into method calls. You can simply use the **`Step Into`** menu item, which will break at the first instruction of the method.

SADE 1.3 Manual Errata

- The dialog box displayed when the target menu is selected is shown as a generic SFGetFile dialog box. In the released product the dialog box contains the line

  ```
  Please select an application or symbol file.
  ```

- In the section describing the different ways to target a program, the manual states that you can drag and drop the icon of the target application on top of the SADE icon. For this feature to work, you may need to rebuild your desktop once.

- The intentional bug introduced in the tutorial program Sample3 can be more destructive than we intended. Running this program on your machine can hang it.

  If it does not hang your system, and you display the stack when the bus error occurs, the output is more informative than documented, in that SADE displays some MacsBug symbols.

- SADE acquired the ability to prompt for source files when needed after the manual was finalized. As a consequence you may never have to use the SourcePath command. This behavior is not documented in the SADE manual, but is described in some detail at the beginning of these release notes.

- SADE now includes an optional menu called Extras which is not documented in the manual, but is described earlier in these release notes.

- Contrary to what the SADE 1.3 Manual states, you can debug Hypercard XCMDs and XFCNs with SADE. The way to do this is described earlier in these release notes.

For SADE Version 1.3

SADE Reference

030-3631-A
Developer Technical Publications

APPLE COMPUTER, INC.

Printed in the United States of America.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables

Preface **About This Manual**

Welcome to SADE®, the Symbolic Application Debugging Environment. SADE is an interactive debugger that can be used with programs written in languages such as Pascal, C, or Fortran, or with programs written in assembly language. SADE can also debug programs written in Object Pascal and C++—with or without MacApp—and it can be used as either a source-level debugger or an assembly-level debugger, no matter what language the target program is written in.

Although SADE is a stand-alone application, its user interface is similar to that of the Macintosh Programmer's Workshop (MPW®). Therefore, if you know how to use MPW, you are already familiar with the look and feel of SADE.

What's in this manual?

This manual describes the most important components and features of SADE, in both reference and tutorial form. Other useful references include the Macintosh Programmer's Workshop Development Environment manual, and, of course, *Inside Macintosh, Volumes I–VI.* This manual includes the following material:

- Chapter 1, "Introduction to SADE," provides an overview of SADE. It includes descriptions of the hardware and software configurations needed to use SADE; an explanation of how to install SADE and how to get started using it; a brief tour of the SADE interface; and a hands-on example showing how to debug a simple program.
- Chapter 2, "Menu Reference," describes the SADE menus and explains in detail the menu commands that are unique to SADE.
- Chapter 3, "Language Reference," describes the syntax and components of the SADE command language and explains how to use the language to debug a program.
- Chapter 4, "C Tutorial Examples," takes you through several C sample debugging sessions.
- Chapter 5, "Pascal Tutorial Examples," takes you through several Pascal sample debugging sessions.
- Chapter 6, "Debugging Special Cases," describes special steps you must take to debug code written using Object Pascal, MacApp, and C++; it also describes how to debug MPW tools.
- Appendix A, "Summary of Commands and Built-in Functions," lists SADE commands and functions according to their uses.
- Appendix B, "Customizing SADE," explains how to modify SADE to meet your specific needs.
- Appendix C, "Formal Grammar," is a formal description of the SADE command language.
- Part II, "Command Reference," lists and describes all SADE commands and built-in functions, in alphabetical order. It describes the syntax and the operation of each command and each function, and presents examples showing how commands and functions are used in SADE scripts.

This manual is organized so you can make the best use of it. If you need an overview of SADE, read Chapter 1; otherwise, you may be able to skim Chapter 1 just to refresh your memory, and then focus on Chapter 2 to learn the SADE menu structure or Chapter 3 to learn more about the SADE command language. Alternatively, you might find that diving into a tutorial is the easiest and quickest way to learn SADE. All the tutorials are short, so you can work through them without investing a great amount of time.

Reference material has been placed in the Appendixes and in Part II so you can find it easily.

Notation conventions

The following notation conventions are used to describe SADE commands:

`courier`	Courier (typewriter) font is used for lines that you must type exactly as shown. (You must also type special symbols—such as `%`, `_`, `†`, and so on—exactly as shown).
variable	Italics are used as place holders; italicized items can be replaced by anything that matches their definition.
[`optional`]	Square brackets are used to enclose elements that are optional.
either | *or*	A vertical bar (|) indicates an either/or choice.
,...	A comma followed by an ellipsis indicates that the preceding item can be repeated one or more times, separated by commas.

Command names are not case-sensitive.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

For more information

APDA® (Apple Programmers and Developers Association) offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple® platforms. You'll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, please contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299

800-282-2732 (United States)
800-637-0029 (Canada)
408-562-3910 (International)
Fax: 1-408-562-3971
Telex: 171-576
AppleLink® address: APDA

Part I **Using SADE**

Chapter 1 **Introduction to SADE**

This chapter introduces some fundamental debugging concepts and describes how SADE® can aid you with the debugging process. It also describes the hardware and software you need and it explains how to install SADE. In addition, it presents a short tutorial that shows how to debug a simple program.

What is SADE?

SADE (Symbolic Application Debugging Environment) is a debugger, that is, a program that can help you debug other programs. Although SADE is modeled closely after the Macintosh Programmer's Workshop® (the SADE editor, for example, is functionally identical to the MPW® editor), SADE is a separate application that runs independently of MPW. You can use SADE to debug applications and MPW tools written in any language supported by MPW, including Pascal, C, C++, Fortran, or assembly language. SADE is compatible with programs written with MacApp.

Hardware and software requirements

To run SADE, you need the following hardware and software:

- a Macintosh Plus computer, a Macintosh SE computer, any computer from the Macintosh II family, or a Macintosh computer that uses future members of the MC68000 microprocessor family.
- at least 2.5 megabytes (MB) of RAM; 4 MB are recommended
- Macintosh system software version 7.0 or later, or Macintosh system software version 6.0 with MultiFinder version 6.1B9.
- MPW version 3.2 or later

SADE supports all MC68000–family microprocessors and the MC68881 floating-point coprocessor. It does not support the MC68851 Memory Management Unit (MMU) or the corresponding MMU registers on the MC68030.

Installation

To install SADE, simply drag the SADE folder from the release disk to any location on your hard disk. The SADE folder does not need to be associated with the MPW folder, although you can place it in the MPW folder if you wish.

△ **Important** If you are using version 6.0.x of the Macintosh system software, you must also replace the MultiFinder program that is in your System Folder with MultiFinder version 6.1B9, which is on the SADE release disk. The MultiFinder program on the SADE release disk contains special code for controlling and accessing the application state. This additional code supports debugging but does not affect your normal use of MultiFinder. To install the special version of MultiFinder, move your current MultiFinder out of the System Folder, copy the new version in, and restart your computer. If you already have MultiFinder version 6.1B9, or if you are running system software version 7.0 or later, skip this step. △

The SADE folder on the SADE release disk contains auxiliary files that must remain in the same folder as SADE in order for the debugger to function properly. These auxiliary files are SADEStartup, SADEUserStartup, SADE.Help, SADE Worksheet, and SADE New User Worksheet.

The SADE folder also contains the file SysErrs.Err, which enables SADE to print error message text. You must place this file in your System Folder.

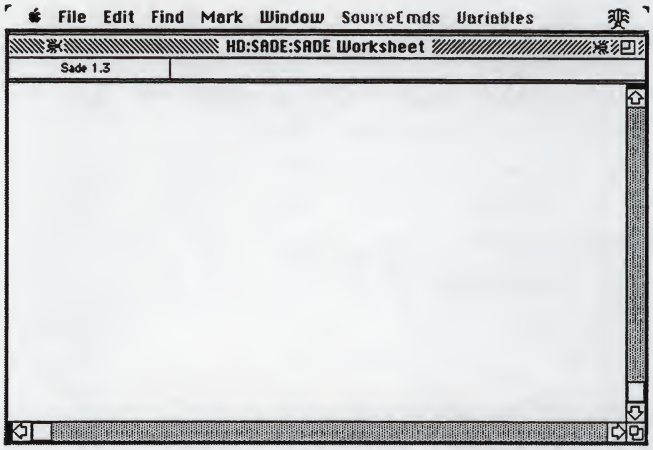
The CExamples and PExamples folders inside the SADE Tutorials folder contain SADE sample applications that you need for working through the tutorials in Chapters 4 and 5. You can place these folders anywhere, but you should not delete them because they are essential for using the tutorials.

The SADE Example Scripts folder contains sample scripts. You can place this folder anywhere.

The SADE user interface

SADE combines a window-based debugging environment, a menu interface, a text editor, and a command language. The menu interface is a standard Macintosh implementation. Figure 1-1 shows the SADE user interface, including the SADE menu bar and the SADE text editor window.

■ **Figure 1-1** The SADE user interface



The SADE menu bar

The SADE menu bar includes editing and file management commands, as well as commands that can be used to control the execution of an application and can display information about the application during a debugging session.

The SADE text editor

With the SADE text editor, you can edit programs by making direct changes in source files. In addition, you can suspend the operation of a program and step through it to see how the program responds to the execution of each statement in its source file. The SADE editor can automatically open special files in separate windows when you want to see the value of a variable at a particular point in a program or when you want to watch the value of a variable change during the operation of a program.

The SADE command language

The SADE command language extends the SADE menu interface by offering you more command options than a menu interface provides. The SADE command language supports structured programming, allows you to create your own debugging functions and procedures, and can be used to write debugging scripts.

Debugging: An overview

As every programmer knows, finding and fixing bugs is an integral part of developing a program. A **bug** is simply a piece of code that causes the program to do something other than what you intended it to do. Debugging a program encompasses three major steps:

- discovering that a bug exists
- finding the code that caused the bug
- fixing the bug

In many cases, detecting the existence of a bug is simple—for example, a program crashes as soon as you launch it, or it freezes when you choose a particular menu command. However, another bug may be more subtle and not so easy to recognize, either because it occurs only under a very limited set of circumstances or because its effect is difficult to spot. For example, a bug might occur only when a particular number of windows are open, or a program might seem to be operating smoothly until you look closely and notice that it is truncating names longer than ten letters.

The next step in the process of debugging, finding the cause of a bug, is usually the most difficult. Your best resource for tracking down the source of a bug is not a particular software tool—although a good tool is invaluable—but your skill as a programmer. Another asset is your knowledge of the application that you are debugging.

Although procedures for determining the source of a bug depend on the kind of bug you encounter, the art of debugging is essentially a matter of narrowing your focus. You begin with an educated guess about the cause of a bug, and then you try to narrow down the amount of code that contains the bug. The next section shows how SADE can help you in this winnowing process.

Once you have discovered a bug and determined its origin, you can go about fixing it—ideally, without introducing any new bugs.

SADE: An overview

How does SADE assist you in the process of tracking down bugs? To begin with, SADE is a **source-level debugger.** It allows you to debug the source-code version of a program in either assembly language or the source language in which the program is written. In contrast, a low-level debugger such as MacsBug displays every program in machine language, no matter what language the program was originally written in. Thus, you have to understand 68000 machine language to use MacsBug, but you don't have to understand assembly language to use SADE.

SADE, SourceBug, and MacsBug

SADE is one of three debuggers offered by Apple Computer, Inc., for use with programs written using MPW. The other two debuggers are MacsBug and SourceBug.

SourceBug, like SADE, is a source-level debugger. SourceBug is designed primarily to debug programs written with MacApp; however, it can also be used to debug programs written in other high-level languages such as C and Pascal.

SourceBug is easier to use than SADE, but lacks some of the most powerful features of SADE. With SourceBug, as with SADE, you can set breakpoints, control a program's execution, and examine the contents of variables. However, you can't change the contents of variables from the SourceBug editor, as you can from the SADE editor, and you can't use SourceBug to write or run debugging scripts. Also, you can't use SourceBug to debug MPW tools; for that you need SADE.

MacsBug is a low-level debugger that can only display a program in machine instructions no matter what language the program was originally written in.

The -sym option

SADE is referred to as a **symbolic debugger** because it uses symbolic information that can be optionally generated by the MPW compilers and the MPW linker when a program is built. SADE uses this information to correlate elements in a program's source code with the way the program behaves when it is executed. Thus, you can use SADE to see exactly how each source-code statement affects the operation of the program.

To include symbolic information in a program, you must use the `-sym on` option (or the `-sym full` option) when you compile and link your program with MPW. When you use the `-sym on` option with the MPW compilation and linking commands, MPW automatically generates a file of symbolic information named *programName*.SYM. SADE cannot work effectively unless such a file exists. For more information about compile commands and the `-sym on` or `-sym full` options, see the Macintosh Programmer's Workshop Development Environment manual and the documentation provided with MPW compilers.

If a program includes some files that have been compiled with symbolic information and others that have not, you can still target the program using SADE. Any time you step into a section of the program that does not have symbol information associated with it, you must operate at the machine instruction level. Since every program compiled by MPW has MacsBug symbols embedded in it, SADE can use the limited MacsBug symbols in this case.

Capabilities of SADE

SADE provides several features that allow you to control the execution of a program:

- setting breakpoints

  A **breakpoint** suspends operation of a program at a specified address or program statement, and passes control to SADE. You can set breakpoints conditionally or unconditionally. A conditional breakpoint suspends operation of a program every time a specified condition is met; for example, a variable is set to a suspect value.

- stepping through a program

  With SADE, you can execute a program one source-language statement or one machine instruction at a time. After each step, SADE highlights the next statement to be executed. You can thus verify that your routines and subroutines are executing in the proper sequence.

  When you step through a procedure, you can either step over routines or into routines called by that procedure. If you are certain that a particular routine is executing properly, you can step over it. On the other hand, if you suspect that a routine contains a bug, you can step into it and examine each of its statements.

- viewing and watching variables

  One way to determine whether your program is executing properly is to verify that routines are passing the proper values to each other and are using variables with valid values. At any time during program execution, you can choose to look at the value of any specified variable. For example, you can set a breakpoint after a program's initialization is complete and then examine certain variables to see if they have been initialized correctly. Alternatively, you can select a **watch variable**—a variable to watch as a program is executing. When you select a watch variable, you instruct SADE to update the variable's value each time SADE is entered.

SADE also provides low-level functionality that allows you to

- examine the heap to see if your application is handling memory properly
- examine the stack to see the call chain
- disassemble an address or address range to see the assembly instructions that your source code is executing
- dump memory locations to see the exact contents of memory areas and display registers
- trace traps to see the Toolbox calls your program is making

Getting started

You can launch SADE by double-clicking on the SADE application icon or on a SADE document icon. Figure 1-2 shows the SADE application icon (on the left), the SADE document icon (in the center), and a SADE symbol file icon.

- **Figure 1-2** The SADE application, document, and symbol file icons

△ **Important** The SADE document icon is used by the Finder™ program to identify any document that is created and saved by SADE—usually a SADE script. Note that the SADE document icon is *not* used to identify programs that are debugged using SADE; programs debugged with SADE are usually created by MPW, and so are identified with the MPW document icon. △

Once launched, SADE opens the SADE **Worksheet window.** This window has no close box and is always present on the screen; otherwise it is just like any other window. You will use it most often to type commands and see the return output. You can also write SADE procedures or functions or keep notes during a debugging session—you can save anything in the Worksheet to another window or file.

As in MPW, you can enter SADE commands from the SADE Worksheet, using the standard MPW editing, selecting, and executing conventions.

To see a list and brief description of all commands, type `Help` and press Enter. To see the syntax of any particular command, type `Help` *commandName* and press Enter. For a complete description of every command, look at Part II of this manual.

Six ways of targeting a program

Targeting means identifying to SADE the application you want to debug. There are six different ways that you can specify the target application for SADE, which are described in the following sections. Five of these methods are manual—the one you choose is a matter of individual preference. The final method described here turns the process around: you put special code in your target application that calls SADE when certain conditions are met.

◆ *Note:* Before you can debug a program in SADE you must compile and link the program successfully.

Double clicking the .SYM file

In the Finder, double click on the .SYM file for your application, which launches SADE, if it is not already running. SADE sets its working directory to the folder containing the symbol file. If it finds an application in the same directory whose name matches the symbol file name, SADE launches the application. If SADE does not find an application whose name matches that of the symbol file, it displays a dialog box asking you to find the corresponding target application.

Double clicking on the symbol file also works for MPW tools. In this case SADE launches MPW if it is not already running. If MPW is not in the same directory, SADE displays a dialog box asking you to locate it.

Dragging the .SYM file onto the SADE icon

In the Finder, drag either the .SYM file for your application or the application itself (or an alias of either) on top of the icon for SADE. The behavior in this case is identical to that of double clicking on the .SYM file.

Selecting the Target menu command

Launch SADE and select the Target menu command from the File menu. You can select the program to target from the dialog box that appears. For more details see the description of the Target menu command in Chapter 2, "Menu Reference."

Using the Target command

Launch SADE and set the proper directory with the following command:

`Directory '`*`HD:YourDirName`*`'`

Then issue the following command to set the target application:

`Target '`*`YourAppName`*`' [using '`*`DifferentSymbolFileName.Sym`*`']`

For details, see the `Directory` and `Target` commands in Part II.

Using the SADEKey

Launch your application. Launch SADE and set the proper directory with the following command:

```
Directory 'HD:YourDirName'
```

You can suspend your running application with the SADEKey combination (Command-Option–keypad period), which breaks the target program the next time a `WaitNextEvent`, `GetNextEvent`, or `EventAvail` routine is called.

Remember that you can not only use the SADEKey to target your application but can use it at any time to halt your program. The SADEKey works if either SADE or your target application is the front most Multifinder application.

Using trap calls

Launch SADE and set the proper directory with the following command:

```
Directory 'HD:YourDirName'
```

Instead of targeting your application manually, you can place trap calls in your application's code that explicitly transfer control to SADE when they are executed.

For example, you might want to debug how your application behaves when you double click on a document created by that application. Or, you might want to halt your program any time it executes code that you did not expect.

Anywhere in your program, add the following trap call.

For C or C++ Programs:

```
#include <Errors.h>

...

SysError(8005);
```

For Pascal Programs:

```
Uses  Errors;

...

SysError(8005);
```

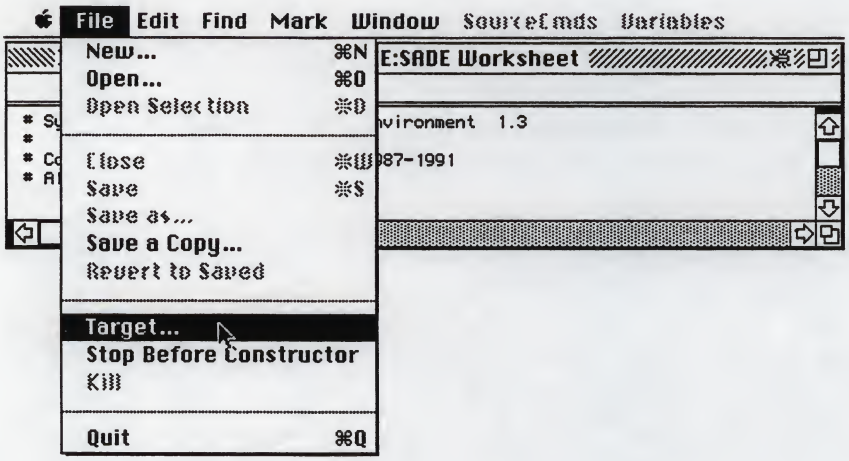
The parameter to the call is a short. Legal values for invoking SADE are in the range 8003 to 32512.

A simple tutorial

The easiest way to learn about SADE is to use it. The SADE distribution disk contains several sample applications and their source and symbol files. This tutorial uses the application, Sample1, which was compiled and linked with the `-sym on` option.

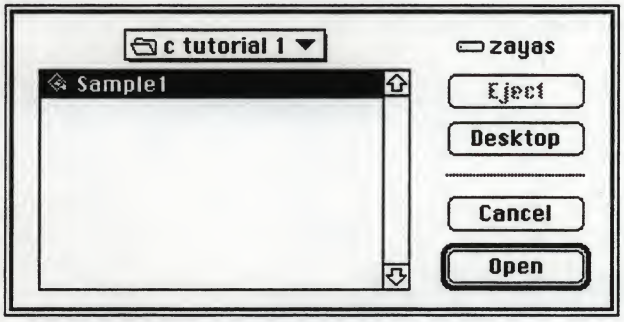
To debug this program, launch SADE, and select Target from the File menu as shown in Figure 1-3.

■ **Figure 1-3** Targeting a program



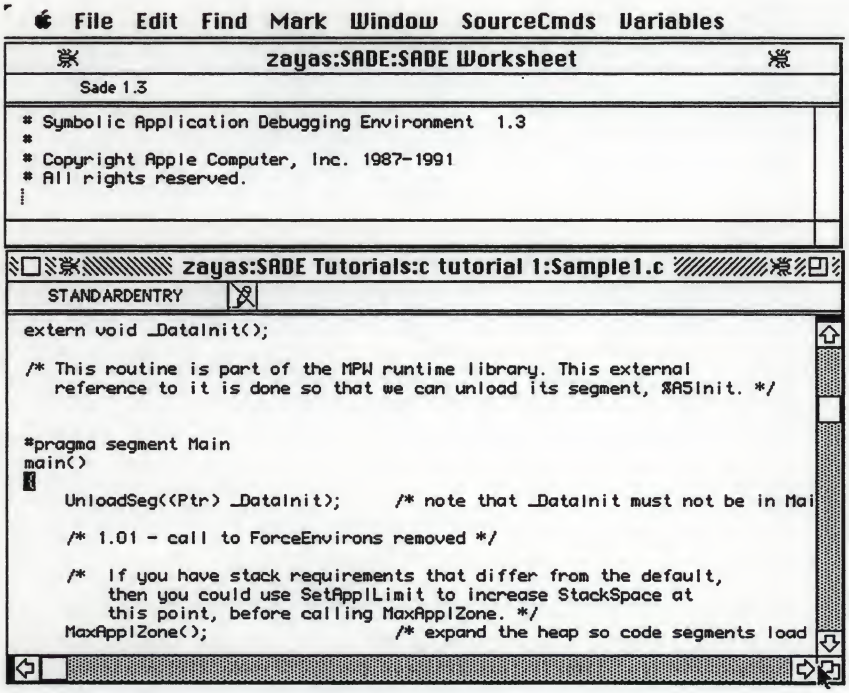
SADE displays a Standard File dialog box asking you to find the application to debug. Navigate through your directory structure to select Sample1, which resides in the directory: SADE Tutorials:CExamples:c tutorial 1. Figure 1-4 shows the Target dialog box.

■ **Figure 1-4** Finding the target program



When you select Sample1, SADE launches the application and stops it at the first line of `main`, as shown in Figure 1-5.

■ **Figure 1-5** A program targeted for debugging



To continue the execution of the program, select Go from the SourceCmds menu. Figure 1-6 shows the SourceCmds menu.

■ **Figure 1-6** Selecting Go

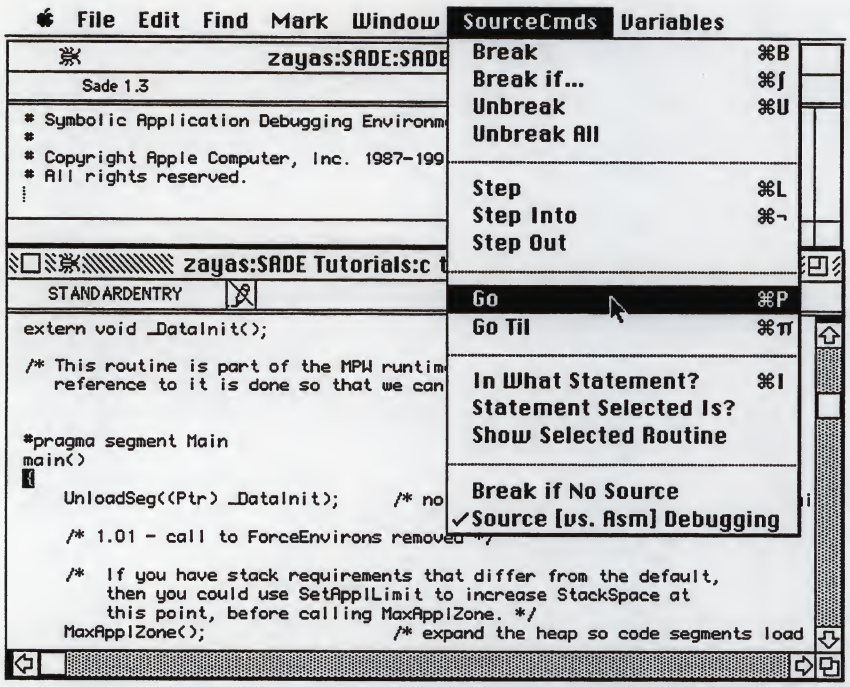
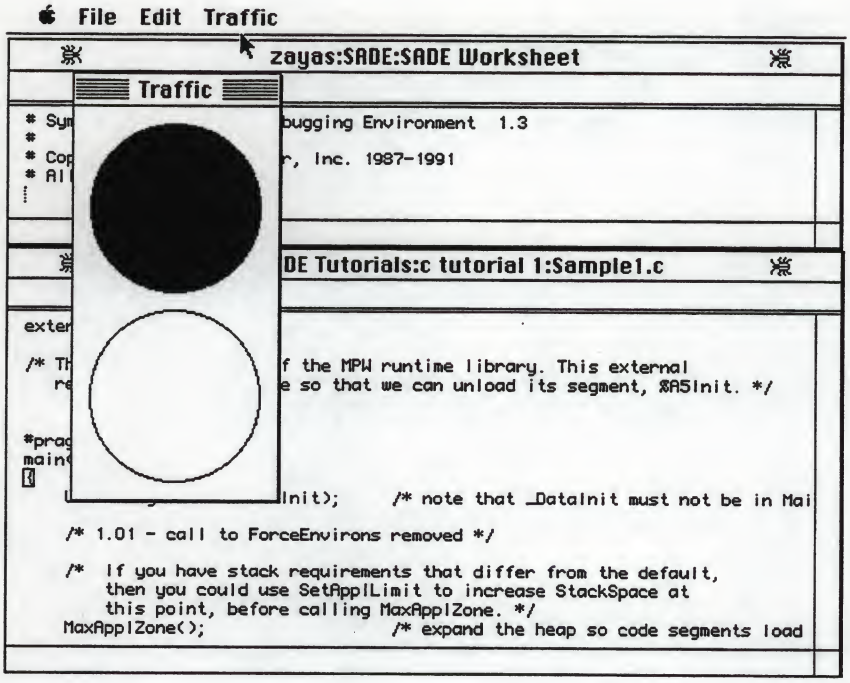


Figure 1-7 shows the sample application and its menu bar, which appear after you select the Go menu command.

■ **Figure 1-7** The running target



Sample1 has one simple function: to switch the traffic light between red and green (if you have a black-and-white monitor, the top circle is darkened when red, and the bottom circle is darkened when green). In versions of the program that do not contain a bug, you can switch the traffic light between red and green by clicking either circle or by choosing the Green Light or Red Light command from the Traffic menu. If you experiment with Sample1, you will quickly discover that you can switch the lights by clicking either circle. The Green Light command works correctly as well. But you will find that the Red Light command in the Traffic menu turns on the green (bottom) light instead of the red one.

Given this observation you may quickly deduce that the bug is in the `DoMenuCommand` routine, so you are going to examine the execution of this routine more carefully. Select `DoMenuCommand` from the Mark menu and SADE displays that routine.

To stop execution of an application in a routine, you place a break point in it. Put the cursor anywhere on the line containing the `switch` statement and select the Break menu command from the SourceCmds menu. Figure 1-8 shows the `switch` statement in the `DoMenuCommand` routine and the SourceCmds menu for setting a breakpoint.

■ **Figure 1-8** Setting a breakpoint

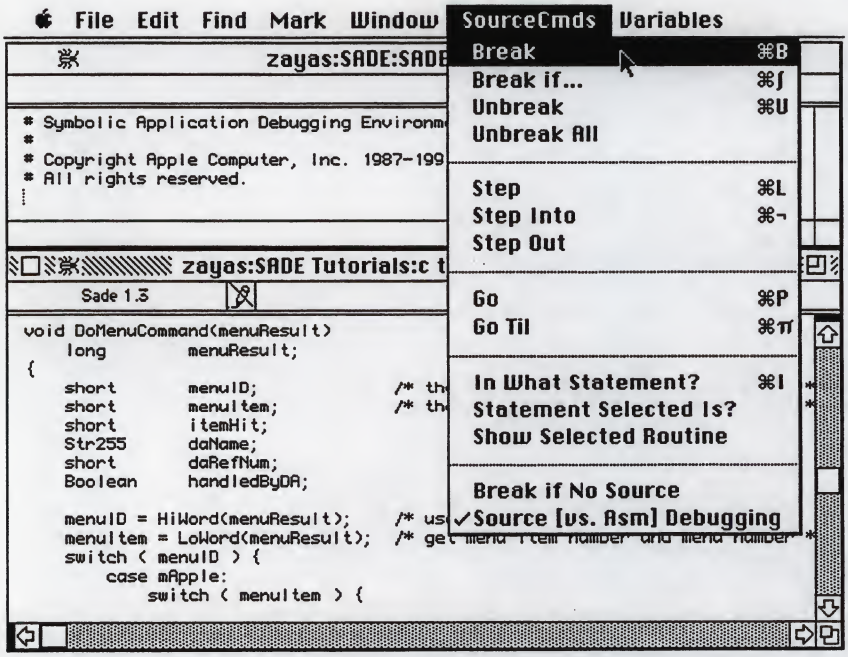
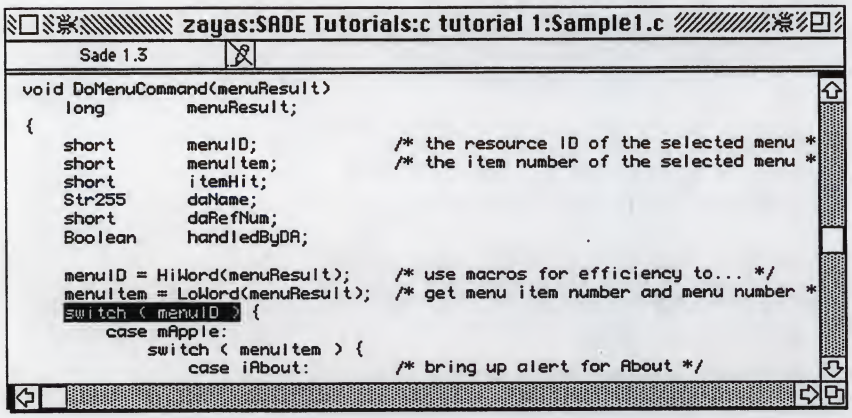


Figure 1-9 shows that when you set a breakpoint, SADE highlights the entire statement in front of which it places the breakpoint.

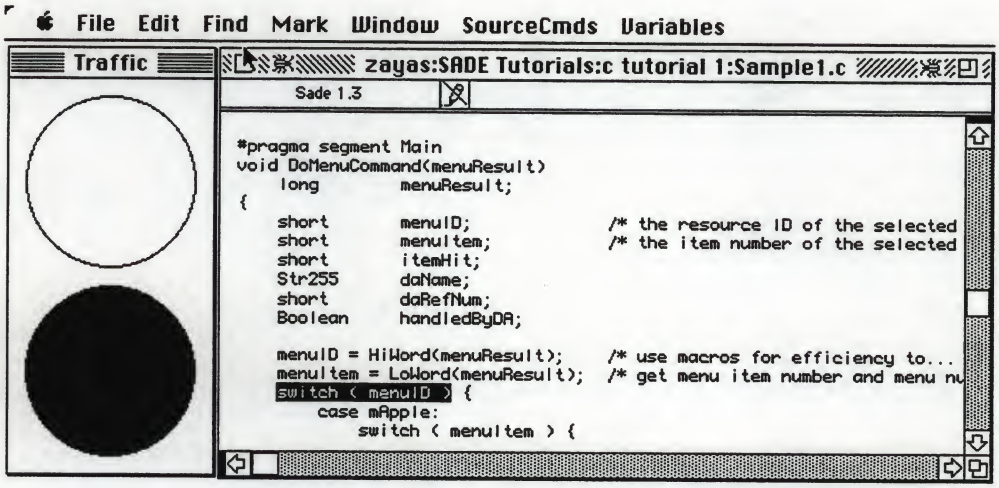
■ **Figure 1-9** After setting a breakpoint



Restart execution of the program by selecting Go from the SourceCmds menu.

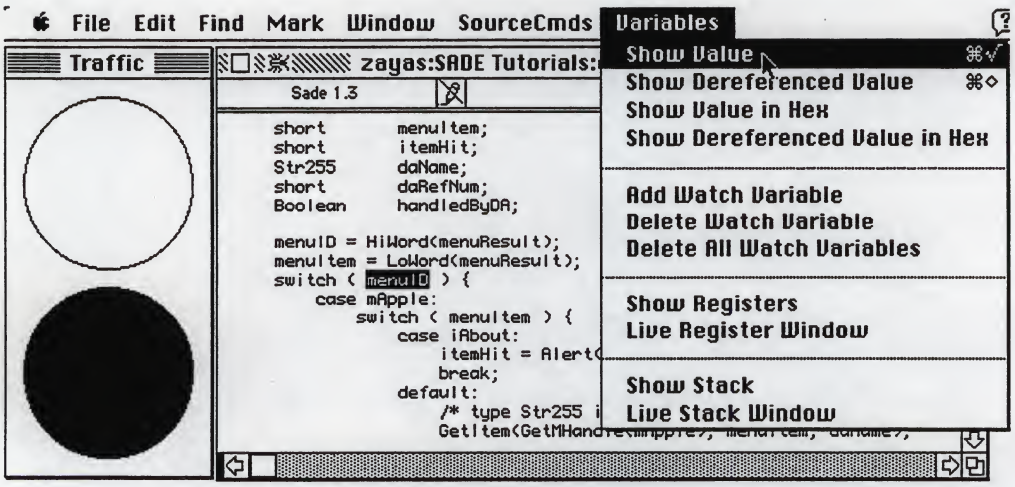
From the Traffic menu select the Red Light menu command. SADE gains control of the program; your display will look something like that in Figure 1-10. (The windows have been rearranged for the sake of clarity). The program has stopped at the `switch` statement on which you set a breakpoint.

■ **Figure 1-10** Halted sample program



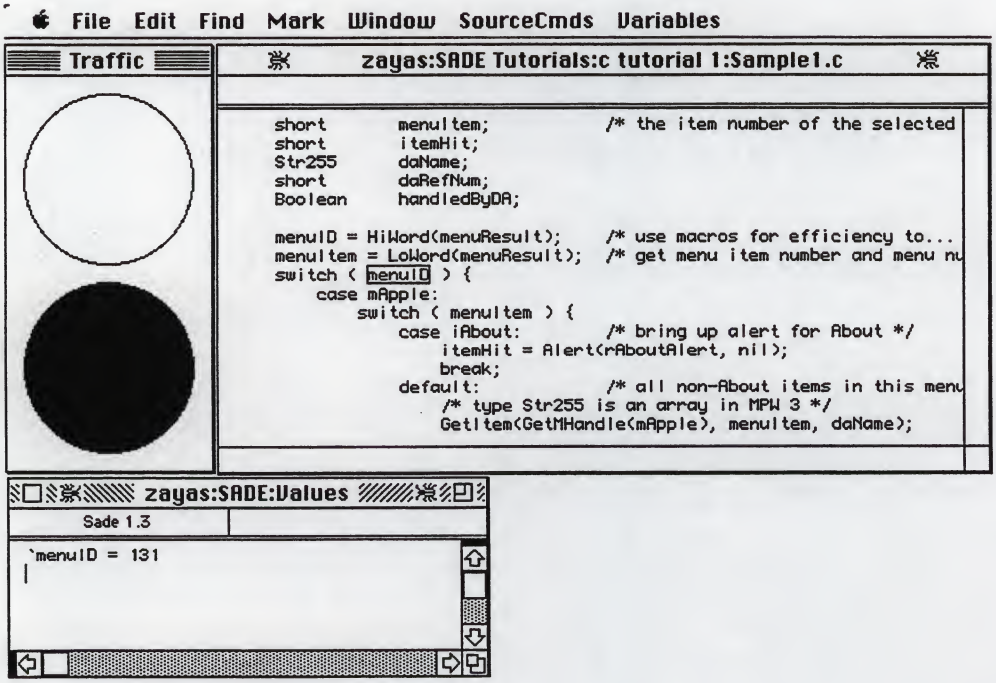
At this point you can examine the value of variables in the target program. For example, to see the value of the variable `menuID`, highlight it, then select Show Value from the Variables menu. Figure 1-11 shows the Variables menu with the Show Value menu command highlighted.

■ **Figure 1-11** Show Value menu



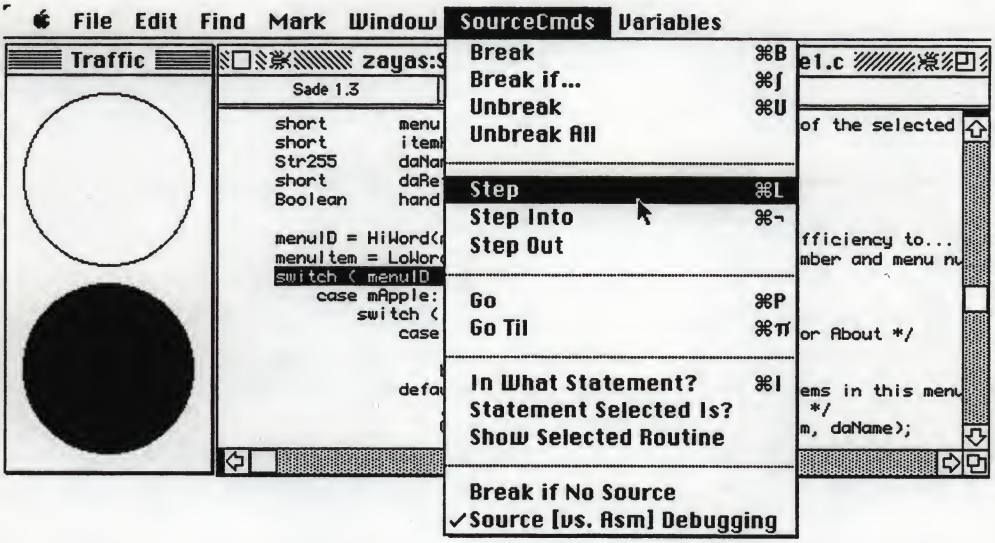
SADE opens up a new window and displays the current value of the variable (131), as shown in Figure 1-12.

■ **Figure 1-12** Value Window



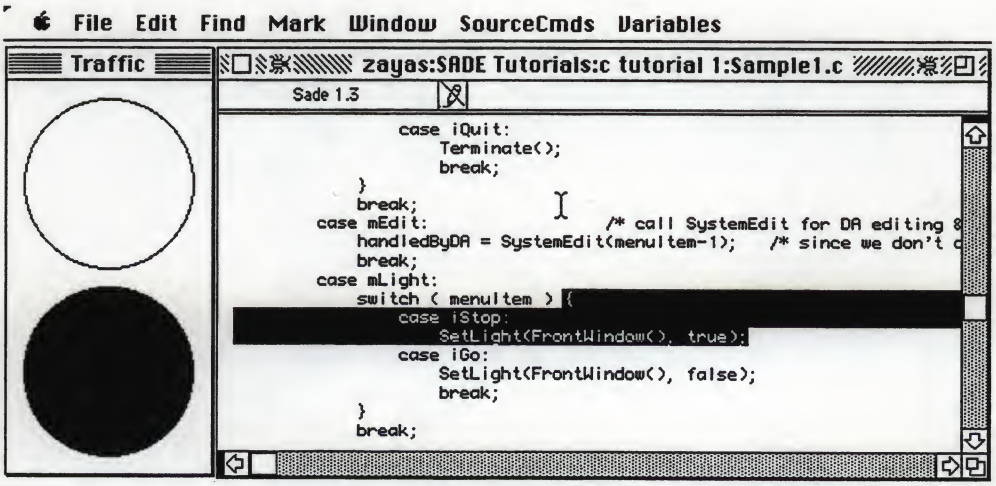
Single step the application to see which one of the case labels the program executes by selecting Step menu from the SourceCmds menu, as shown in Figure 1-13.

■ **Figure 1-13** Single Stepping



Single step again. The display should look like that in Figure 1-14.

■ **Figure 1-14** Discovering the bug



At this point the tutorial has accomplished its purpose by familiarizing you with the SADE basics. The bug in the program, which you may have already guessed, is that there should be a `break` statement after the text that is highlighted in Figure 1-14. For a more complete analysis of this bug and a more in depth look at SADE's capabilities, see C Tutorial 1 in Chapter 4, "C Tutorial Examples."

To quit SADE select Quit from the File menu or press Command-Q. As a part of its exit action, SADE terminates the stopped application.

Where to go from here

Now that you have used SADE to debug a simple program, read Chapter 2, "Menu Reference," to get a detailed and exhaustive description of all the menu commands. You may be able to operate SADE effectively with just that.

However, SADE also provides a scripting language to automate repetitive debugging tasks. To learn to use the SADE command language, read Chapter 3, "Language Overview."

There are more samples in Chapter 4, "C Tutorial Examples," and Chapter 5, "Pascal Tutorial Examples." The sample debugging sessions in these chapters demonstrate additional debugging techniques and give examples of how to use the SADE command language.

Chapter 6, "Special Debugging Cases," tells how to target an MPW tool and gives tips about debugging programs written with MacApp, C++, or Object Pascal.

After you have familiarized yourself with how SADE works, you can customize it to suit your own needs. You can

- add, change, or delete menus and menu commands
- define macros to execute SADE commands or command sequences
- customize the way SADE behaves when it takes control from an application, and what happens when you quit SADE

See Appendix B, "Customizing SADE," for details.

Chapter 2 **Menu Reference**

This chapter provides an overview of the SADE menus. It describes in detail the SourceCmds menu, the Variables menu, and commands in other menus that are unique to SADE.

Many users will find that they are able to operate SADE using only the menu commands described in this chapter. However, SADE also provides a rich scripting language for tackling complicated or repetitive tasks. After the description of each menu command in this chapter is its SADE command language equivalent.

This chapter assumes that you are familiar with the MPW menu structure. Thus, the only commands in the File, Edit, Find, Mark, and Window menus that this chapter describes are those that differ from their MPW counterparts. Refer to the Macintosh Programmer's Workshop Development Environment manual for complete descriptions of the standard menus.

Using menus and menu commands

SADE is written in accordance with the Apple Human Interface Guidelines, so you'll find its menu interface familiar and comfortable if you are an experienced Macintosh user. However, many SADE menu commands are scriptable—that is, they have corresponding commands (described in Part II, "Command Reference") that you can execute interactively on command lines or use non-interactively in scripts. Commands that you can issue from command lines and scripts as well as by choosing menu commands are identified in the menu descriptions in this chapter.

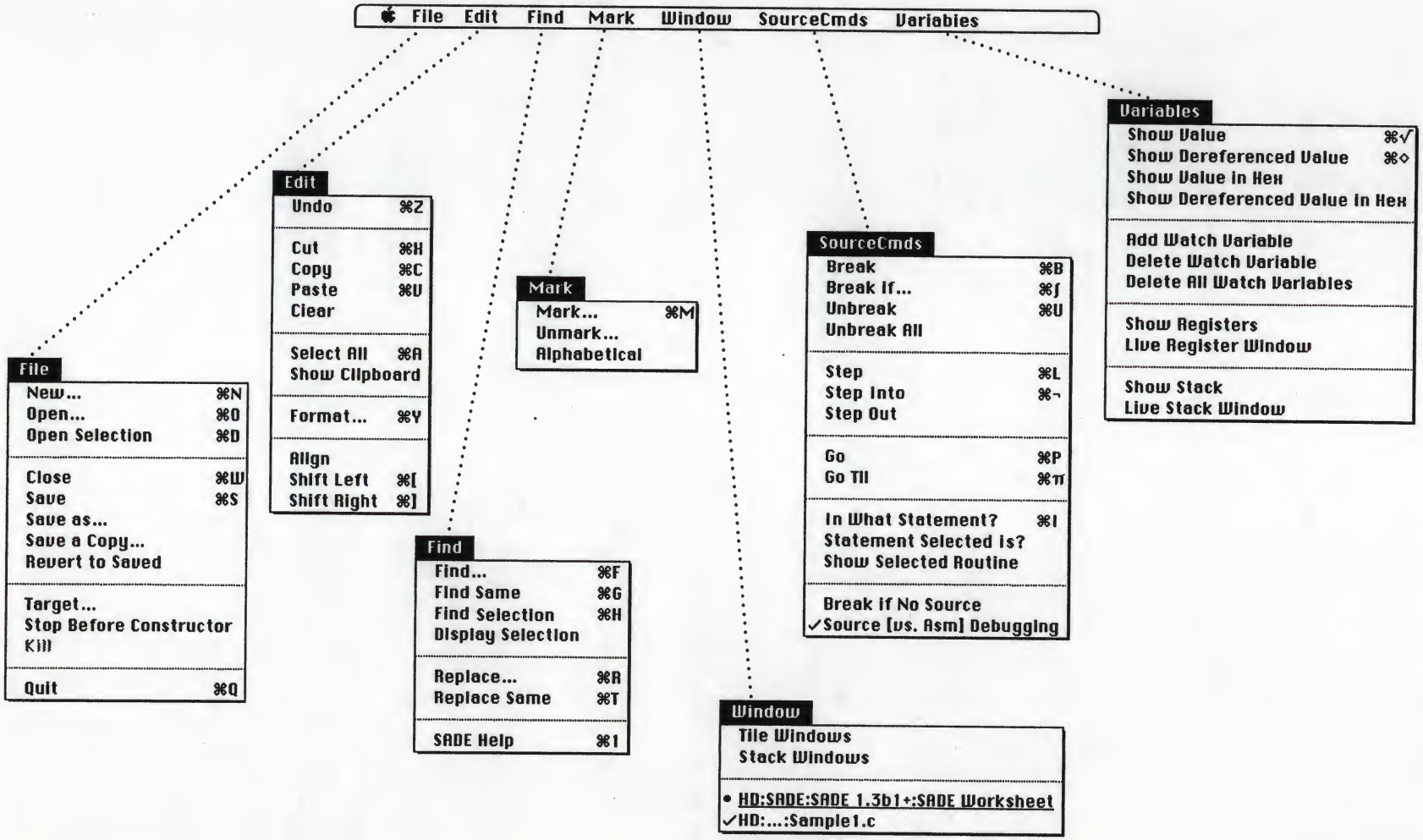
Most SADE commands work with statements or variables that you select in a program's source file or in the SADE Worksheet. In some cases you must select the appropriate text by highlighting it. For example, to examine the value of a variable, you must highlight the variable in your source text and then choose the Show Value command from the Variables menu. On the other hand, to set a breakpoint on a statement you need only click anywhere on the line and then select the Break command from the SourceCmds menu.

This chapter uses the term *highlight* when you must select the text; it uses the term *click* when placing the insertion point anywhere on the appropriate text is sufficient.

SADE menu bar

Figure 2-1 shows the SADE menu bar, a standard Macintosh menu interface. The six menus on the left—labeled , File, Edit, Find, Mark, and Window—have the same names as the six menus that occupy the same positions on the MPW menu bar. In addition to these six menus, SADE has two menus that MPW does not have: the SourceCmds menu and the Variables menu.

■ **Figure 2-1** The SADE menus



The File menu

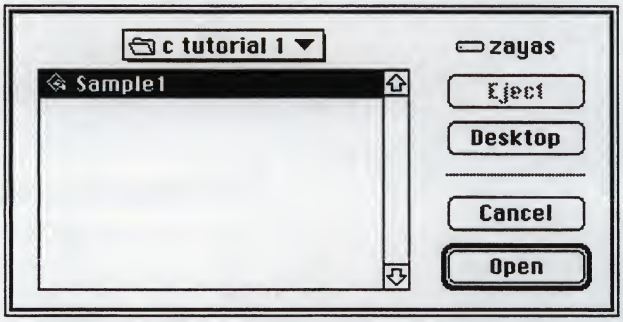
With the File menu, you can manage and manipulate files. SADE has three File commands that are not included in the MPW File menu: Target, Stop Before Constructor, and Kill.

Target

You can use the Target command in the File menu to target, or select, a program that you want to debug using SADE. When you choose the Target menu command, SADE displays a dialog box like the one in Figure 2-2. You can then select the correct directory, if necessary, and select a program to target for debugging.

If SADE does not find a symbol file with the same name as the target application (*target*.SYM), SADE opens a dialog asking you to locate the symbol file. SADE assumes that your source files are in the last directory that you selected.

- **Figure 2-2** The Target dialog box



The Target command not only identifies a program to debug, but it also launches the program, stopping at the first statement of function `main` in a C program or the main program in Pascal. SADE also highlights the statement at which it has stopped. To continue the execution of the program, you can perform one of the following tasks:

- choose Go from the SourceCmds menu
- use the keyboard equivalent, Command-P
- issue the `Go` command from the SADE Worksheet window

There is a scriptable `Target` command; to use it, you must first select the correct directory by executing the `Directory` command. Then you can target a program in that directory by entering the `Target` command in the SADE Worksheet. For more information about the `Target` command, see Part II, "Command Reference."

Stop Before Constructor

The Stop Before Constructor menu command determines how SADE behaves when it opens a target program. Normally when SADE opens a target program, it runs the program, stopping at the first statement of the user code. If you select the Stop Before Constructor menu command, SADE interrupts execution of the target program before any user code is executed. At this point you can set a breakpoint in your static constructor and resume the target program.

This menu command is only useful for C++ programs containing class static objects that have user-defined constructor code associated with them.

The scriptable equivalent of this command is to set the SADE variable `StopBeforeStatic` to 0 or 1.

Kill

When you have started debugging a program, the Kill menu command becomes enabled in the File menu. The Kill menu command terminates the target program and ends the current debugging session.

If the target program is an MPW tool, the Kill menu command terminates the tool and leaves MPW running. Thus, MPW becomes the target. To terminate the debugging of an MPW tool and start debugging an application, you must execute the scriptable SADE command `Untarget`.

There is a scriptable `Kill` command, described in Part II, "Command Reference."

Quit

The Quit menu command saves all temporary files and terminates SADE. If the target program is suspended SADE terminates it; otherwise the target is released.

The keyboard equivalent for the Quit menu command is Command-Q.

The scriptable equivalent of the Quit menu command is to issue the commands:

```
Save all; Quit
```

The Find menu

The only command in the SADE Find menu that works differently from its MPW counterpart is the SADE Help command which displays help about SADE.

SADE Help

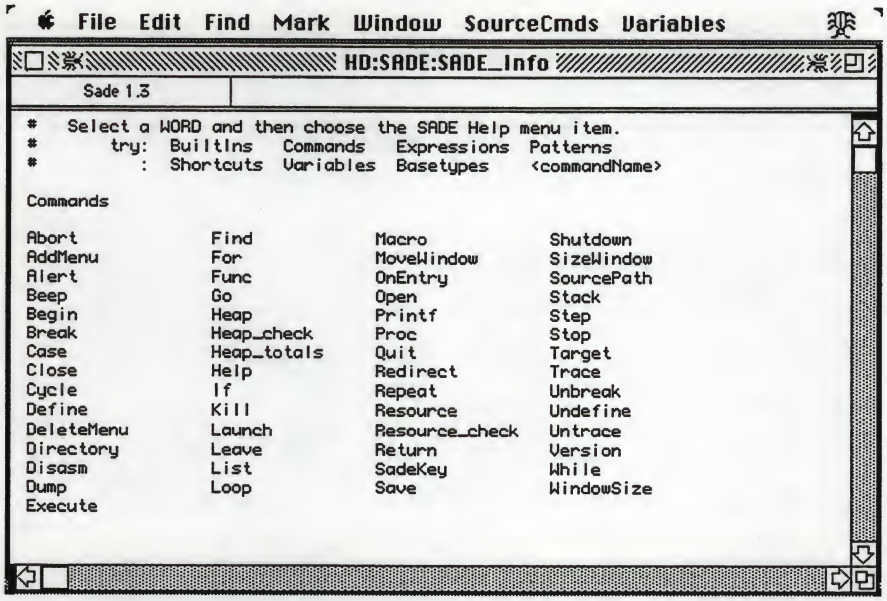
With the SADE Help menu command, you can obtain information about any SADE command, any built-in SADE function, and any of the following:

- expressions
- patterns
- shortcuts
- variables
- base types

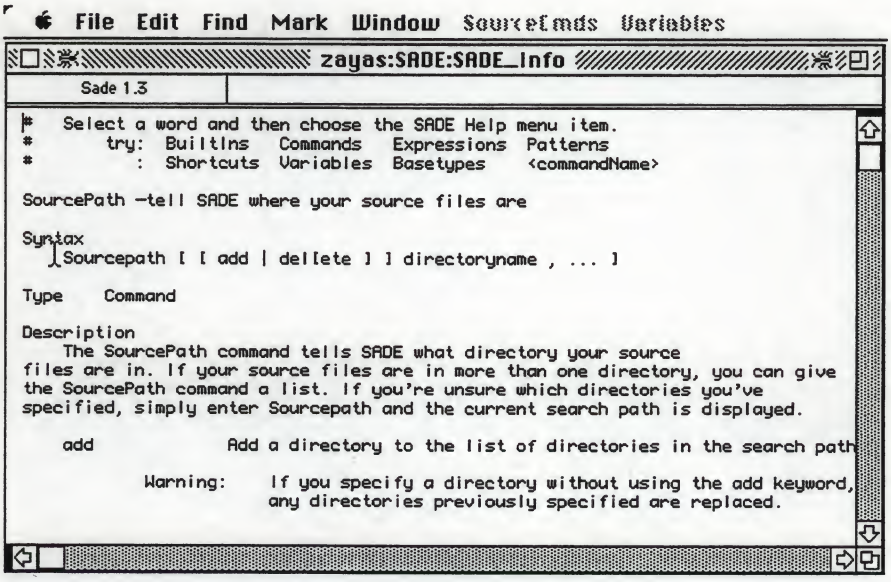
You can use the SADE Help command in either of these ways:

- To see what kind of help SADE offers, make sure that nothing is highlighted in the active window, and then choose the SADE Help menu command. SADE responds by opening a special window called SADE_Info, shown in Figure 2-3. In the SADE_Info window, SADE lists all help topics that are available—that is, all SADE commands and all built-in functions.
- In either the SADE_Info window or the SADE Worksheet window, highlight (or type and then highlight) any topic for which help is available. Then choose the SADE Help menu command. If the SADE_Info window is not already open, SADE opens it. In the SADE_Info window, SADE then displays help about the topic selected, as shown in Figure 2-4.

■ **Figure 2-3** Help topics listed in the SADE_Info window



■ **Figure 2-4** Help text displayed in the SADE_Info window



The keyboard equivalent for the SADE Help menu command is Command-1.

The scriptable `Help` command does not open the SADE_Info window but displays the requested information in the Worksheet window.

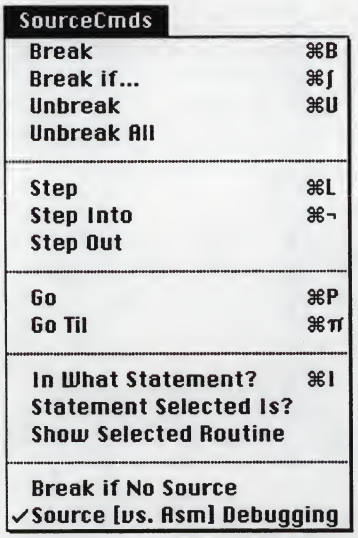
The SourceCmds menu

The SourceCmds menu, shown in Figure 2-5, provides commands that allow you to

- set and remove breakpoints (including conditional breakpoints)
- step through a program (including stepping over, into, and out of subroutines)
- resume execution of the program
- navigate the source files for the program
- step through a target program in either the source language or in assembly language, and display the program on the screen in either the source language or assembly language
- control how SADE behaves when there is no source information for the program counter

◆ *Note:* Before choosing any of the commands in the SourceCmds menu, you must target a program to be debugged by issuing the Target command. If a program has not been targeted, the SourceCmds menu is dimmed and cannot be used.

- **Figure 2-5** The SourceCmds menu



Break

The Break command in the SourceCmds menu sets a breakpoint on a statement containing the current selection. The source file must be the active window.

You can set a breakpoint immediately after targeting a program or when the program has been suspended. You can set as many breakpoints as you wish in the target program. When you run a program after setting a breakpoint, SADE interrupts the program just before the statement with the breakpoint is executed. When the program halts, SADE brings the source file to the front and highlights the statement containing the breakpoint.

Breakpoints remain set after you resume program execution. To remove a breakpoint, you can

- highlight all or part of the statement containing the breakpoint and then choose the Unbreak command from the SourceCmds menu, as described later in this chapter
- choose the Unbreak All command from the SourceCmds menu, as described later in this section
- execute the `Unbreak` command or the `Unbreak All` command from a command line or a script, as described in Part II, "Command Reference"

You must set a breakpoint somewhere after the start of a routine—that is, on or after the opening brace of the routine (in C) or on or after the `BEGIN` keyword (in Pascal); otherwise, SADE returns a message stating that it cannot determine the break address from the available source information.

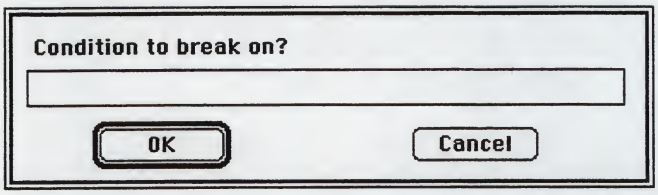
The keyboard equivalent for Break is Command-B.

There is also a scriptable `Break` command, described in Part II, "Command Reference."

Break If...

The Break If command sets a conditional breakpoint. To set a conditional breakpoint, select the source statement on which to set the breakpoint and choose Break If. When the Break If dialog box appears, you can type in a conditional statement.

■ **Figure 2-6** The Break If dialog box



When you run a program after setting a conditional breakpoint, SADE interrupts the program if the condition you have specified is met. SADE then brings the program's source file to the front and highlights the source statement containing the breakpoint. If the condition is not met, SADE does not interrupt the program.

For example, if you type

```
menuID = 0
```

in the dialog box, then the next time the specified statement is executed, the program will stop if and only if the variable `menuID` equals 0.

One use of a conditional breakpoint is to define a condition that should not occur if a program is running properly. For example, to test whether a program can identify all of the menus that a user might choose, you can put a breakpoint in the routine that handles menus and specify the break condition as `menuID = 0`. When you run the program and click a menu, if the program does not know that you have clicked a menu or if it cannot identify the menu, the program assigns 0 to `menuID`. At that point, the program breaks because the condition you specified has been met—and you know that you have a problem. Otherwise, the program continues to run without interruption and you know that your menu-handling routine can identify each menu in the program.

Breakpoints remain set after you resume program execution. To remove a breakpoint, you can select the statement with the breakpoint and choose Unbreak from the SourceCmds menu. Alternatively, you can execute the `Unbreak` command from a command line or a script, as described in Part II.

If you set a conditional breakpoint using a variable that is out of scope, SADE displays a warning because it cannot verify that the condition will be meaningful within the context of the breakpoint.

The keyboard equivalent for Break If is Command-Option-B (Command-∫).

There is also a scriptable `Break` command, described in Part II, that allows you to create conditions on which to break.

Unbreak

The Unbreak command removes a breakpoint from a selected statement. You can use the `List break` command, described in Part II, to see the statements on which breaks are set.

To remove a breakpoint, select all or part of the source statement from which you want the breakpoint removed, and then choose Unbreak from the SourceCmds menu. If you attempt to remove a breakpoint from a statement that does not contain one, SADE does not warn you. However, if you try to remove a breakpoint from code that is outside a statement (for example, a routine name), SADE returns a message stating that it cannot determine the break address from the available source information.

The keyboard equivalent for Unbreak is Command-U.

There is also a scriptable `Unbreak` command, described in Part II.

Unbreak All

The Unbreak All command removes all the breakpoints that have been set in the target program.

The scriptable equivalent of the Unbreak All menu command is `Unbreak all`, described in Part II.

Step

The Step command executes the target program one source-language statement or one machine instruction at a time, beginning at the next instruction in the program—that is, at the address specified by the current value of the program counter. Before you choose the Step command, execution of the program must be suspended. Operation of a program is suspended when it is first targeted and when it encounters a breakpoint or other exceptional conditions.

If the Source [vs. Asm] Debugging menu command is enabled (the default setting), SADE steps through the target program by executing one source-language statement at a time. After each step, SADE highlights the next statement in the source file to be executed. If the Asm [vs. Source] Debugging menu command is enabled, the Step command executes one machine instruction at a time, and displays the next machine instruction to be executed.

The Step command treats traps and subroutines as single statements—that is, it steps over them. If you wish to step through a called subroutine, use the Step Into menu command. (Note that Step Into also treats traps as single instructions.)

▲ **Warning** Don't try to step over a routine that does not return to the caller—for instance, `longjump()`. SADE steps over procedure and function calls by setting the trace bit until after the `JSR` or `BSR` is executed. After the call is made SADE replaces the return address with the address of a SADE routine. Since routines like `longjump()` restore a previously saved register set, including a new stack pointer, the return address saved by SADE is lost. If you want to go to the routine restored by `longjump()`, you must repeatedly choose `Step Into` (with assembly-language debugging selected) until the registers have been modified, and then do a source step. Better still, if you know which routine the jump will go to, set a breakpoint in that routine. ▲

The keyboard equivalent for Step is Command-L.

There is also a scriptable `Step` command, described in Part II. It enables source or assembly debugging and stepping over and into subroutines.

Step Into

The Step Into menu command operates exactly as the Step menu command unless it encounters a subroutine call instruction, in which case it steps into the called routine. When the target program executes a subroutine call, SADE highlights the first statement in the called routine.

Before executing the Step Into command, you must first suspend the execution of the targeted program, either by pressing the SADE Key combination (Command-Option–keypad period) or by setting and encountering a breakpoint.

If the Source [vs. Asm] Debugging menu command is enabled (the default setting), SADE steps through the target program by executing one source-language statement at a time. After each step, SADE highlights the next statement in the source file to execute. If the Asm [vs. Source] Debugging menu command is enabled, the Step command executes one machine instruction at a time, and displays the next machine instruction to be executed.

If you want to step over subroutines, use the Step menu command. If you have stepped into a subroutine and want to step out, use the Step Out menu command.

The Keyboard equivalent for Step Into is Command-Option-L.

The scriptable `Step` command, described in Part II, includes an option for stepping into subroutines.

Step Out

The Step Out command takes you out of a subroutine. If you have stepped into a subroutine that you don't want to step through, choose Step Out. SADE returns you to the calling routine and highlights the statement following the subroutine call.

Go

The Go command resumes execution of the target program beginning at the address specified by the current value of the program counter.

The keyboard equivalent for Go is Command-P.

There is also a scriptable `Go` command, described in Part II.

Go Til

The Go Til command sets a temporary breakpoint on the statement you have selected in the source file and resumes program execution. When the program encounters the breakpoint, SADE is reentered and the breakpoint is removed.

The keyboard equivalent for Go Til is Command-Option-P.

The scriptable `Go` command, described in Part II, includes an option for setting a temporary breakpoint.

In What Statement?

The In What Statement? command displays and highlights the source statement that corresponds to the instruction at the address specified by the current value of the program counter. That is, when the program is suspended, In What Statement? highlights the next source statement to be executed. Suppose, for example, that the target program has encountered a breakpoint and is suspended in a particular routine. You have scrolled through the source file (maybe to set another breakpoint or just to look at a different routine). You can now return to the point where SADE suspended your program using In What Statement? command.

The keyboard equivalent for In What Statement? is Command-I.

The scriptable equivalent of In What Statement? is `AddrToSource(PC)`.

Statement Selected Is?

The Statement Selected Is? command displays the location of a statement selected in a source file using a SADE identification label. A SADE identification label is made up of the name of a function or a procedure and an index number, combined in this format:

procedureName.(*n*)

where *procedureName* is the name of a procedure or a function, and *n* is an index number. The index number 0 is used to represent the beginning of a procedure or a function. The first statement in a procedure or a function has index number 1, the second statement has index number 2, and so on. For example, the fifth statement in a procedure or function called `DoMenuCommand` has this identification label:

```
DoMenuCommand.(5)
```

You can use identification labels for many purposes in SADE—for example, to set a breakpoint at a specific statement in a program. For example, this command sets a breakpoint at the fourth statement in a routine named `DoMenuCommand`:

```
Break DoMenuCommand.(4)
```

The scriptable equivalent of the Statement Selected Is? menu command is the built-in function, `SourceToAddr`, which returns the name and offset of the statement selected in the target window.

Show Selected Routine

The Show Selected Routine command opens the source file and highlights the statement whose name (and index number) you have highlighted in the SADE Worksheet. This command also allows you to find and display source code if you know its address.

You may want to use the Show Selected Routine command in conjunction with the `Stack` command to examine the source code for a particular point in the calling chain. For example, suppose you have targeted the Sample1.c program and have set a breakpoint at the first statement of the `SetLight` routine, which is

```
if (newStopped != gStopped) {
```

When the program suspends execution at this statement, you can examine the calling chain by choosing the Show Stack command from the Variables menu or executing the scriptable `Stack` command. In this case, the Show Stack or `Stack` command generates this output:

```
Frame At  Owned By                Called By
$004FEC90 main                    %__MAIN+$003C
$004FEC88 EventLoop               main.(5)
$004FEC58 DoEvent                 EventLoop.(15)+$0004
$004FEC34 DoContentClick          DoEvent.(18)+$0004
$004FEC20 SetLight                DoContentClick.(1)+$0012
```

The second column in the last line of this output shows that the statement at which the program has halted is somewhere in the `SetLight` routine. The third column of the last line shows that the `SetLight` routine is called by the first statement in the `DoContentClick` routine. That statement is referred to in the example by its SADE identification label, `DoContentClick .(1)`. If you highlight the label `DoContentClick .(1)`, and select the Show Selected Routine command in the SourceCmds menu, SADE opens Sample1.c and highlights the line

```
SetLight (window, !gStopped);
```

in the `DoContentClick` routine.

The scriptable equivalent of the Show Selected Routine menu command is to use the built-in function `AddrToSource`, which can highlight a source statement when you provide the statement's name and index number or a valid code address.

Break If No Source

The Break If No Source command determines how SADE behaves when you step into a section of your program for which SADE has no symbol information. When Break If No Source is not checked (the default), a Step Into command continues stepping until there is source statement information available for the instruction at the program counter. The benefit of this is that SADE does not break in out-of-line arithmetic routines such as ULMODT and display mysterious lines of assembler. A further benefit is that Object Pascal and MacApp method dispatches are treated as indivisible operations: stepping into a method call will break at the first instruction of the method.

The down side of not setting Break If No Source is that if SADE steps into something without statement information, it will appear SADE has hung when in fact it is just executing your code two to three orders of magnitude slower than normal. This down side should only affect you if you compile some of your code with symbols and some without and then inadvertently step into the code without symbols.

When you choose the Break If No Source menu command, a check mark appears beside the menu item. In this case, SADE breaks when it encounters code for which no source information is available.

The scriptable equivalent of this menu command is setting the value of the SADE variable `BreakIfNoSource` to 1 or 0.

Source [vs. Asm] Debugging

The Source [vs. Asm] Debugging command toggles between source-level and assembly-level debugging. Source-level debugging is the default setting and the menu item reads "Source [vs. Asm] Debugging." When assembly-language debugging is in effect, the menu item reads "Asm [vs. Source] Debugging."

The Source [vs. Asm] Debugging command affects the Step and Step Into commands in the SourceCmds menu. When source-level debugging is enabled, the Step commands execute one source statement at a time. When assembly-level debugging is enabled, the Step commands execute one machine instruction at a time.

The Source [vs. Asm] Debugging command does not affect the behavior of the scriptable `Step` command, which can be used with options that allow you to specify source-level or assembly-level debugging.

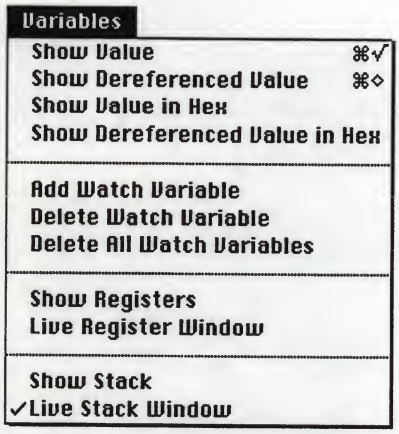
The Variables menu

The Variables menu, shown in Figure 2-7, contains commands that can

- display the values of variables (including dereferenced values and hexadecimal values)
- display the contents of microprocessor registers
- add and delete watch variables
- examine the stack to trace the calling chain of any routine in a target program

◆ *Note:* Before choosing any of the commands in the Variables menu, you must target a program to be debugged by issuing the Target command. If a program has not been targeted, the Variables menu is dimmed and cannot be used.

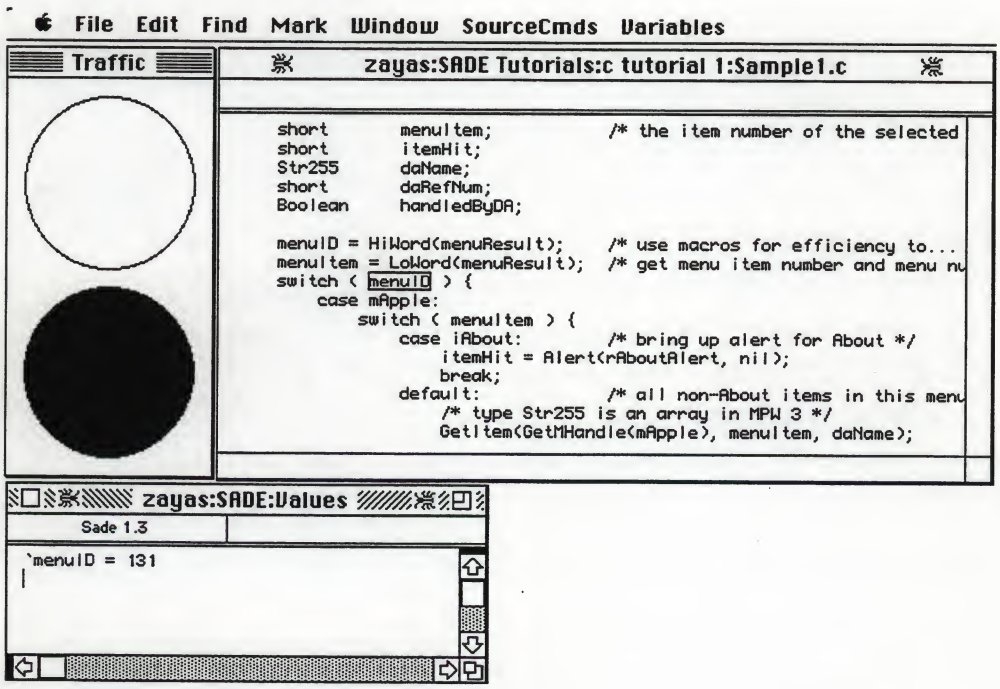
■ **Figure 2-7** The Variables menu



Show Value

The Show Value command displays the name and value of any variable that is highlighted. Before you choose Show Value, you must highlight a variable in any window. SADE then opens a window named Values. In the Values window shown in Figure 2-8, SADE displays a line showing that the current value of the variable `menuID` is 131.

■ **Figure 2-8** The Values window



In the Values window, SADE displays this warning message if the variable is undefined or outside the scope of the current program counter:

varName `= Undefined/Out of scope`

By default, SADE displays numeric values in the Values window as decimal values. You can obtain values in hexadecimal notation by choosing the menu command Show Value in Hex (or Show Dereferenced Value in Hex).

If the value of a variable changes during the execution of a program, SADE does not update the Values window by overwriting the variable's old value with the variable's new value. If you want to watch the value of a variable change as you step through a program, you can choose the Add Watch Variable menu command.

A backquote character before the variable name identifies it as a program symbol instead of a SADE variable or system symbol. See Chapter 3, "Language Overview," for a detailed discussion of name spaces.

The keyboard equivalent for the Show Value menu command is Command-Option-V.

There is no scriptable version of the Show Value command, but you can obtain the value of any variable by simply typing its name and pressing the Enter key. In addition, you can use the `Printf` command to display the value of a variable in various formats, and also to convert values to different data types. For example, SADE displays numeric values as decimal by default, but the `Printf` command can display them as hexadecimal, binary, octal, or floating-point values. For example, this command prints the value of the variable `menuID`:

```
Printf "%d", menuID
```

For more information about the `Printf` command, see Part II, "Command Reference."

Show Value in Hex

The Show Value in Hex command displays the name and hexadecimal value of any variable highlighted in a targeted program. The Show Value in Hex menu command is identical to Show Value except that it displays the value in hexadecimal notation rather than decimal notation.

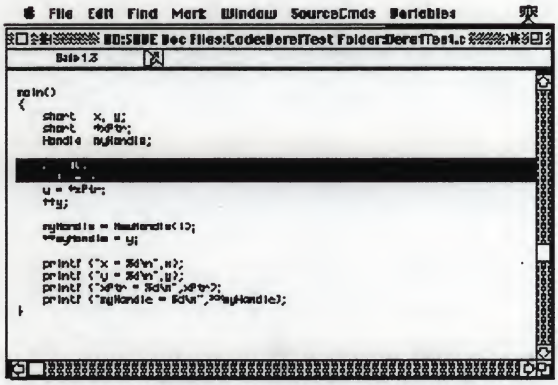
If you attempt to obtain a value that is not of integral type by selecting Show Value in Hex, SADE displays a warning message.

The scriptable equivalent of Show Value in Hex is the SADE `Printf` command with the `x` or `X` option.

Show Dereferenced Value

The Show Dereferenced Value command displays the dereferenced value of any pointer variable highlighted in a target program. (The dereferenced value of a pointer variable is the value pointed to by the pointer variable.) Before you choose Show Dereferenced Value, you must highlight a variable in the source file of the target program. SADE then opens a Values window like the one shown in Figure 2-8. In the Values window, SADE displays the selected variable and its dereferenced value. A caret following the variable name shows that the variable's dereferenced value is displayed. For example, suppose that a program contains a variable named `x` and a pointer named `xPtr`. Also assume that the value of `x` is 10, and that `xPtr` points to `x`. These values are set in the highlighted lines of C code in Figure 2-9.

■ **Figure 2-9** Values of variables



Suppose that a target program contains the fragment of code shown in Figure 2-9, and has suspended execution at a breakpoint in the last line of code on the screen:

```
printf ("myHandle = %d\n",**myHandle);
```

If you now highlight the variable `xPtr` and select the menu command Show Dereferenced Value, this line is displayed in the Values window:

```
`xPtr^ = 10
```

There is no scriptable command for showing the dereferenced value of a variable, but SADE provides two operators—^ (for Pascal-style syntax) and * (for C-style syntax)—that you can use to obtain the dereferenced value of a variable. In fact, if your program uses handles (pointers to pointers), or even more levels of indirection, you can use the pointer operator as many times as necessary to follow the chain of pointers. For example, if you highlight the symbol `myHandle` in Figure 2-9 and then choose Show Dereferenced Value, this line is displayed in the Values window:

```
`myHandle^ = ^Byte($0027A104)
```

This line indicates that `myHandle` is a handle that points to a pointer at address $0027A104. The pointer which `myHandle` points to, points to a value of type `Byte`. To obtain the dereferenced value of the pointer pointed to by `myHandle`, you can simply select the symbol `myHandle^` in the Values window and choose Show Dereferenced Value again. SADE then displays the doubly-dereferenced value of `myHandle` in the Values window:

```
`myHandle^^ = 11
```

The keyboard equivalent for the Show Dereferenced Value command is Command-Option-Shift-V.

- *Note:* If you find this key combination (or any other key combination) difficult to press, you can change the key mapping to a different combination (see the SADE New User WorkSheet for an example).

Alternatively, you can double-dereference `myHandle` by typing `myHandle^^` or `**myHandle` in the SADE Worksheet window, and pressing Enter.

Show Dereferenced Value in Hex

The Show Dereferenced Value in Hex command displays, in hexadecimal notation, the dereferenced value of any pointer variable highlighted in a targeted program. The Show Dereferenced Value in Hex command is identical to the Show Dereferenced Value command except that it displays the value in hexadecimal rather than decimal notation.

If you attempt to obtain a value that is not of integral type by selecting Show Dereferenced Value in Hex, SADE displays a warning message.

Add Watch Variable

The Add Watch Variable command displays the value of a selected variable in the target program and updates the value each time SADE is entered. Before you choose the Add Watch Variable command, you must highlight a variable in any window. SADE then opens a window, named Variable Watch, that lists the current values of all variables you want to watch during a debugging session. You can select as many as 10 watch variables.

Each time SADE is entered (for example, after a step) SADE updates all watch variables.

If the variable is outside the scope of the current value of the program counter, SADE displays an out-of-scope message instead of the variable's value:

varName `= Undefined/Out of scope`

The Add Watch Variable command does not maintain a record of the values of watch variables. Each time SADE updates the Variable Watch window, the previous value of each watch variable is overwritten with the variable's new value.

◆ *Note:* See the NewUserWorksheet for a scriptable way of keeping a running record of a watch variable.

Delete Watch Variable

The Delete Watch Variable command removes a selected variable from the list of watch variables that are updated and displayed as the target program runs. Before you choose Delete Watch Variable, you must highlight a variable (in any window) that has been chosen as a watch variable.

Delete All Watch Variables

The Delete All Watch Variables command removes all variables from the list of watch variables displayed in the Variable Watch window.

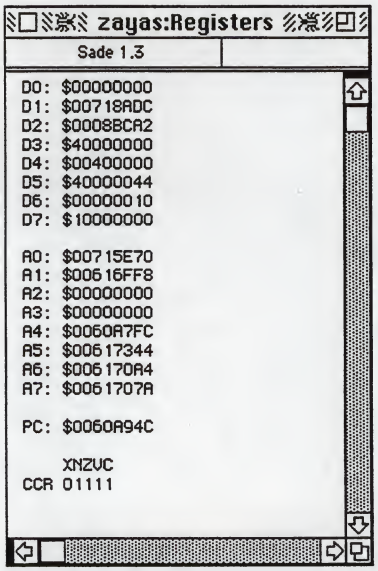
Display Registers

The Display Registers command opens a window named Registers in which SADE displays the current values of these microprocessor registers:

- the data registers (D0–D7)
- the address registers (A0–A7)
- the program counter (PC)
- the condition code register (CCR)

The Registers window is shown in Figure 2-10.

■ **Figure 2-10** Values of registers



The Display Registers command does not automatically update the values of the microprocessor registers as you step through a program. If you want to see updated values, you can choose the Live Register Window command from the Variables menu.

Live Register Window

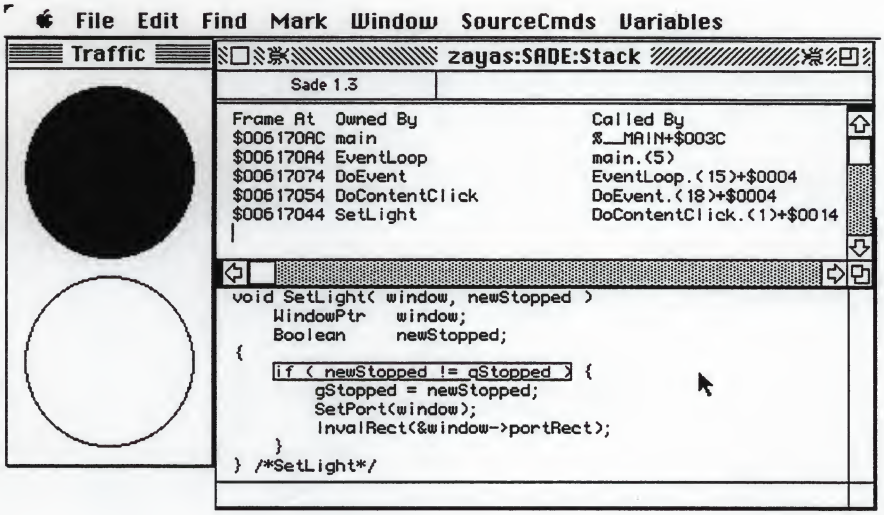
When you select the Live Register Window menu command (a check mark appears by its name), SADE updates the Registers window each time SADE is entered (for example, after a step).

The registers whose values are displayed by the Live Register Window command are the same as those displayed by the Register Display command.

Show Stack

The Show Stack command opens a window titled Stack and displays the calling chain of the target program. The stack window is shown in Figure 2-11.

■ **Figure 2-11** The Stack window



The Show Stack command does not automatically update the values stack display as you step through a program. If you want to see updated values, you can choose the Live Stack Window command from the Variables menu.

When you examine a program's calling chain using the Show Stack menu command or the `Stack` command, and no SADE symbols are available, SADE lists routines using their MacsBug symbols.

There is a similar scriptable `Stack` command but it displays its output in the SADE Worksheet window rather than in a special Stack window. See the description of this command in Part II, "Command Reference."

Live Stack Window

When you select the Live Stack Window command (a check mark appears by its name), SADE updates the Stack window each time SADE is entered (for example, after a step).

Chapter 3 **Language Overview**

You do not have to use the SADE command language to perform most kinds of common debugging operations, but it can increase the capabilities of SADE tremendously. This chapter introduces the SADE command language and explains how it works. Special attention is given to how SADE handles expressions, symbols, strings, and operators.

How SADE interprets commands

This section explains how SADE's interpretation of commands—using full expression evaluation—differs from that of MPW.

It also explain how SADE treats three execution commands—`Go`, `Target`, and `Step`—differently from other commands.

Full expression evaluation

SADE interprets command lines using full expression evaluation, a method of evaluating everything on the command line before it executes the line, in much the same way that a compiler evaluates a command when it is compiling a program. (MPW Shell, in contrast, evaluates commands in seven steps, as explained in the Macintosh Programmer's Workshop Development Environment manual.)

In evaluating a command line, SADE interprets each word that is not enclosed in single or double quotation marks as a **symbol.** Symbols include components of the SADE command language, symbols recognized by the Macintosh Toolbox or Operating System, and symbols defined in the target program. For more information about symbols, see the section "About Symbols" later in this chapter.

If SADE encounters a word on a command line that is not enclosed in quotation marks and is not one of the symbols SADE recognizes, SADE returns an error message. Just as in a programming language such as C or Pascal, you must enclose strings, such as filenames, pathnames, and directory names in a matched pair of single or double quotation marks. For instance, to open a file named sample1.c in the current directory, you could type

```
open 'sample1.c'
```

or

```
open "sample1.c"
```

As another example, you could target an application named Sample1 by issuing a pair of commands in this way:

```
Directory 'HD:SADE:SADE Tutorials:CExamples:C Tutorial 1'
Target 'sample1'
```

- *Note:* You could also target the Sample1 program, of course, by selecting the Target command from the File menu.

Differences between SADE and MPW

SADE does not support all of the features provided by the MPW command language. The following features are supported by MPW but not by SADE:

- regular expressions
- the conditional execution operators `||` and `&&` used between commands (SADE does recognize these operators *within expressions* as the logical operators `AND` and `OR`)
- the input/output symbols >, >>, ≥, ≥≥, <, Σ and ΣΣ
- the pipe symbol |

SADE does provide a special command—`Redirect`—that you can use to redirect output to a specific window. For example, suppose that you typed and entered these three commands in the SADE Worksheet window, either in a block or one at a time:

```
Open 'TestWindow'
Redirect 'TestWindow'
'"Hello"'
```

The `Open` command in the first line of this example opens a SADE window called TestWindow, the `Redirect` command in the second line redirects all SADE output to TestWindow, and the third line prints the word *Hello* in TestWindow. From that point on, all SADE output appears in TestWindow. (For SADE's purposes, the words *window* and *file* are synonymous, since SADE, like MPW, opens all windows as text files.)

◆ *Note:* When you have executed the example, you can stop directing output to TestWindow and revert to the previous output file by executing the command `Redirect Pop`.

SADE execution commands

In SADE, as in MPW, you can execute multiple commands simultaneously, either by selecting them as a block of text and then pressing the Enter key or by placing them in a script file and then executing the script using the `Execute` command. Ordinarily, SADE executes commands in the order in which they appear on a command line or in a script. However, SADE treats `Go`, `Target`, and `Step` —known as **execution commands**— differently from other commands.

When SADE executes a command line or a script containing multiple commands, and one of those commands is an execution command, SADE evaluates the execution command last. Therefore, unless you take special steps to control the order in which SADE commands are executed, you can put *only one execution command* on a command line or in a script that contains multiple commands to be executed simultaneously.

- *Note:* If a script or a block of commands contains more than one execution command, the results are unpredictable.

If you want to place more than one execution command in a script or on a command line, you can use a special option—`onEntry`—which SADE provides for each execution command. (SADE also uses the `onEntry` keyword as the name of a command; for more information on the `onEntry` command, see Part II, "Command Reference.") By using the `onEntry` option following an execution command, you can specify what actions you want SADE to take after the execution command is completed. Thus, you can use `onEntry` to carry out multiple execution commands at one time.

The use of the `onEntry` option is illustrated in this SADE script:

```
Directory 'HD:SADE:SADE Tutorials:CExamples:C Tutorial 1'
Target 'Sample1' onEntry ∂
Begin
      Go Til main.(0) onEntry Begin
            Printf "Application targeted and at statement 0.\n"
            Printf "PC is at %P\n", where (Δpc)
            Step onEntry Begin
                  Printf "Application is at statement 1.\n"
                  Printf "PC is at %P\n", where (Δpc)
                  Kill
                  Printf "Application is killed.\n"
            End
      End
End
```

In this example, `onEntry` is used three times: after the `Target` command, after the `Go Til` command, and after the `Step` command. Each time `onEntry` is used, it is followed by a definition of actions that are to be taken the next time the associated execution command is encountered in the target program.

For more information about the `onEntry` option, see the `Go`, `Target`, and `Step` commands in Part II, "Command Reference."

About symbols

SADE recognizes three classes of symbols:

- **SADE symbols,** which can be user-defined or defined by SADE; these symbols include commands, procedures, functions, macros, and SADE variables
- **program symbols,** which are defined in the target program; these symbols include procedures, functions, and global and local variables
- **system symbols** such as Toolbox traps, microprocessor registers, and low-memory globals

These three classes are also referred to as name spaces. You can display the value of a symbol by entering its name in the Worksheet window. SADE then evaluates the symbol and displays it according to its type.

Distinguishing between program and system symbols

SADE provides two operators that you can use to distinguish between program symbols and system symbols:

- the backslash accent character (`) indicates a program symbol
- the delta character (Δ—Option-J) indicates a system symbol

For example, the abbreviation PC is a system symbol for the Macintosh microprocessor program counter. Therefore, you can obtain the value of the program counter by typing and entering the letters PC preceded by the Option-J (Δ) symbol, as follows:

```
ΔPC
```

How symbol evaluation works

When SADE evaluates a symbol, it first checks to see whether the symbol is a SADE symbol. If it is not, SADE tries to identify it as a program symbol. If that attempt is also unsuccessful, SADE tries to identify the symbol as a system symbol. If all three identification attempts fail, SADE returns an error message.

Because of the order in which symbols are evaluated, a SADE symbol can mask a program symbol or a system symbol of the same name, and a program symbol can mask a system symbol of the same name.

As an illustration of how symbol evaluation works, suppose a target program has a program variable named `PC`. As noted in the previous section, `PC` is also the system symbol for the program counter. In a case such as this, the program symbol `PC` masks the system symbol `PC`, so you cannot obtain the value of the program counter by entering the system symbol `PC`. However, you can always obtain the current value of the program counter by entering: `ΔPC`.

In addition to avoiding masking, the ` and Δ operators speed up the search process. So it can be advantageous to use them even in programs that have no conflicting symbol names.

To avoid name collisions, most of the SADE variables defined in the SADEStartup file have two underscore characters in front of their names. According to ANSI language standards, user programs are not supposed to contain names starting with __.

Case sensitivity

When evaluating symbols, SADE first performs a case-sensitive search. If it doesn't find the symbol, SADE converts all of the characters in the symbol's name to uppercase and searches again. Pascal programmers can use the `Case` command to turn off case sensitivity, as explained under the command's listing in Part II, "Command Reference". SADE then converts all symbols to uppercase before any search, speeding up the search process.

The first character of a symbol identifier must be an uppercase or lowercase letter (A–Z or a–z), an underscore character (_), or a percent sign (%). Subsequent characters can be letters, digits (0–9), underscores (_), dollar signs ($), number signs (#), percent signs (%), or "at" symbols (@). You can make other characters part of an identifier by preceding them with an escape character (∂, produced by Option-D) or a backslash character (\) and enclosing them in double quotation marks. Symbol names may be any length, but only the first 127 characters are significant. Specific compilers can have smaller limits.

MacsBug symbols

Although SADE needs the symbolic information produced by the `-sym on` or `-sym full` option in order to be used most effectively, SADE does not warn you if you target a program that has been compiled and linked without the `-sym on` or `-sym full` option. Thus, if an application includes some files that have been compiled with symbolic information and other files that have not, you can target the program and use SADE to debug the files that do have symbolic information. In fact, you can even debug portions of the program that lack SADE symbol information by using MacsBug symbols.

Every program compiled under MPW contains MacsBug symbol information embedded in its object-code file. This is true whether or not the program has been compiled using the `-sym on` or `-sym full` option. You can tell SADE that you are searching for a MacsBug symbol by preceding the symbol with the μ (Option-M) character. Whenever SADE displays a MacsBug symbol, it precedes the symbol with μ. If you use the μ character to search for a MacsBug symbol in a module that has no SADE symbol information, it doesn't slow down the search process.

When you examine a program's call chain by using the Show Stack menu command (described in Chapter 2, "Menu Reference") or the `Stack` command (described in Part II, "Command Reference"), SADE lists routines by using their MacsBug symbols if no SADE symbols are available.

Program symbols

SADE enables you to identify two primary types of program symbols:

- **program variables,** which you can examine to obtain information about the target program's state at different points in its execution
- **source statements,** which you can use to control program operation—for example, to set breakpoints

- *Note:* A variable reference in SADE always refers to a variable's value, not its address. You may think that because you can specify a breakpoint with the name of a procedure, you can assign the name of a procedure to the program counter (the CPU PC register) as a way of restarting the program at that point. For example, suppose you make the following assignment:

  ```
  PC := DoMenuCommand
  ```

  When you run the program, SADE returns an error message similar to this one:

  ```
  Error number 28 at $408061B2 in "sample1"
            408061B2  2038 0162        *MOVE.L     $0162,D0
  ```

  To recover from this error you must terminate the target program (with the `Kill` command), target it again, and then continue with your debugging session.

  The error message means that SADE cannot find the source code indicated by the program counter. If you had checked the value of the PC register before running the program, you would have found that its value was 0.

  The reason for this error is that you must assign an address, not a value, to the program counter. You can do that by using `@DoMenuCommand` rather than simply `DoMenuCommand`, as the address operator, in the statement that assigns a value to the program counter:

  ```
  PC := @DoMenuCommand
  ```

  Of course, if you know the hexadecimal value for the address of `DoMenuCommand` you can use that as the address operator. For example:

  ```
  PC := $2AD242
  ```

Simple references to symbols

In a command line or a script, the simplest way to refer to a program variable is by a using a **simple reference.** A simple reference is simply the name of a symbol—for example, you can type the name of a variable and press Enter to see its value:

```
newStopped
```

You can also use the `Printf` command, which enables you to format the output. For example, this command writes the current value of a variable named `newStopped` to the Worksheet:

```
Printf "%d", newStopped
```

Simple references can be used only to access symbols that are within the **current name scope**—that is, in the scope that exists where execution of the program has stopped. If execution stops inside a particular function or procedure, two kinds of variables are within the current name scope and thus can be accessed using simple references:

- the program's global variables
- any variable defined in the procedure in which execution has stopped

Partially qualified symbol references

If a symbol is outside the current name scope of a routine in which execution has stopped, it is sometimes possible to access the symbol using a partially qualified symbol reference. SADE can access a symbol using a partially qualified reference if it can deduce the location of the symbol using the information provided in the partial reference. The syntax of a partially qualified symbol reference is:

procedure1[.*procedure2*...][.*variable*]

This construction has three elements:

procedure1 — The name of the function or procedure in which execution has stopped.

In Pascal, the *procedure1* element can be the name of a procedure or a function. In assembly language or C, *procedure1* is the name of a function.

procedure2 — (Optional.) A function or procedure that lies within the function or procedure identified by *procedure1*. A period (.) separates the *procedure1* element from the first *procedure2* element that follows it. If more than one *procedure2* element is used, each *procedure2* element is set off from the previous *procedure2* element by a period.

Each *procedure2* element represents one level in a target program's calling sequence. You can use zero or more levels of procedure name qualification: no levels of qualification when accessing unit level variables, one level of qualification when accessing first-level procedures, and more levels of qualification when accessing nested procedures. (Nested procedures are meaningful in Pascal but not in C.)

variable — (Optional.) A simple or structured variable reference. The period character (.) before the *variable* element delimits it from the preceding *procedure2* element.

In Chapter 4, "C Tutorial Examples," in the section C Tutorial 2, there are examples of references to partially qualified symbols in actual programs. For example, the following is a partially qualified reference that appears in that tutorial.

```
GlobalFunc.StackAllocated
```

Fully qualified symbol references

If you want to obtain information about a program symbol and a partially qualified reference does not provide SADE with enough information to find the symbol, you must provide SADE with a fully qualified symbol reference. You need to use a fully qualified reference when you want to obtain information about a symbol that is defined in a unit, PROGRAM statement, or compilation unit other than the one in which program execution is suspended.

A fully qualified symbol reference has this syntax:

unit [.*procedure*...][.*variable*]

This construction has three elements:

unit

The unit name. The backslash character (\) that precedes *unit* shows that the reference is fully qualified. If no backslash precedes *unit*, SADE interprets *unit* as a procedure name.

In Pascal, *unit* is either the name of a unit or the name of a PROGRAM statement, if that is the main program unit. In assembly language or C, the unit name is the name of the compilation unit; that is, the filename without a source-code extension (.c or .a). You must precede special characters in the filename with the escape character (∂).

procedure

(Optional.) A procedure or function that lies within the unit, PROGRAM statement, or compilation unit identified by *unit*. A period (.) separates the *unit* element from the first *procedure* element that follows it. If more than one *procedure* element is used, each *procedure* is set off from the previous *procedure* by a period.

Each *procedure* element represents one level in a target program's calling sequence. You can use zero or more levels of procedure name qualification: no levels of qualification when accessing unit level variables, one level of qualification when accessing first-level procedures, and more levels of qualification when accessing nested procedures. (Nested procedures are meaningful in Pascal but not in C.).

This syntax also allows referring to multiple activation records of the same function on the stack.

variable (Optional.) A simple or structured variable reference. The period character (.) before the *variable* element delimits it from the preceding *procedure* or *unit.*

To refer to a compilation unit global variable when program execution is suspended in a different unit, you can use the unit name without a procedure name to identify it. For example, this command displays the value of the global variable `gBossyRover` in the compilation unit Moof even if program execution is suspended inside a different compilation unit:

```
Printf "%d", \Moof.gBossyRover
```

References to statements

In a partially or fully qualified symbol reference, you can identify a statement within a procedure by using its SADE identification number. A SADE identification label has this format:

procedureName.(*index*)

where *procedureName* is the name of a procedure or a function, and *index* is an index number. The index number 0 is used to represent the beginning of a procedure or a function, and statements within the procedure or function have consecutive numbers beginning with 1. The first statement in a procedure or a function has index number 1, the second statement has index number 2, and so on. For example, the fifth statement in a procedure or function called `DoLoop` has this identification label:

```
DoLoop.(5)
```

You can use SADE identification labels as arguments to SADE commands. For example, this command sets a breakpoint at the fourth statement in a routine named `DoMenuCommand`:

```
Break DoMenuCommand.(4)
```

To refer to a statement in a procedure in which execution is suspended, you can omit the *procedureName* part of the reference and use the `.(`*index*`)` form by itself, in this format:

```
.(5)
```

In a partially qualified reference, this is the format for using a SADE identification label to identify a statement within a function or a procedure:

procedure1[.*procedure2...*][*statement*.(*index*)]

In a fully qualified reference, this is the format:

unit[.*procedure...*][statement.(index)]

These constructions are the same as the constructions given earlier in this section for partially qualified and fully qualified symbol references, except that in each of these new cases, the *index* element replaces the *variable* element in the previous construction:

index (Optional.) The SADE identification label which SADE uses to identify statements in source files. The beginning of a procedure or function is labeled *procedureName.*(0). The first statement in a procedure or function is *procedureName.*(1), the second statement is *procedureName.*(2), and so on.

In SADE, the name of a procedure or function by itself refers to the starting point of the procedure's code. In a SADE identification label, index number .(0) corresponds to the entry point into a routine before the routine's `LINK` instruction is executed. At this time, the stack frame has not been set up, and the procedure's local variables and parameters do not yet have meaningful values. If you wish to check the value of a parameter, you must go to index number .(1); in other words, this identification label is the entry point to `DoMenuCommand` routine:

```
DoMenuCommand.(0)
```

At this point in the routine, the routine's local variables and parameters are not defined.

Step once and you are at this point,

```
DoMenuCommand.(1)
```

where the stack frame is set up and local variables and parameters are valid.

As noted earlier in this chapter, you can use SADE identification labels as arguments to SADE commands. For example, this command sets a breakpoint at the fourth statement in the routine `DoMenuCommand`:

```
Break DoMenuCommand.(4)
```

In this case, the `Break` command accepts a SADE identification label in the same way it accepts an address. However, if you want to assign the address of a procedure to a register, you should precede the name with the `@` operator, as in `PC := @temp`. The value of a procedure is a special code reference that includes information about resources and offsets, while the `@` operator specifies a simple address at a specific point in time.

Referencing structured types

So far, this section has described simple variables, like `menuID`, that are of the SADE basic types, such as `Integer`. SADE also supports structured types common in high-level programming languages, using the following operators:

`.` `->`	*var*`.`*name* or *var*`->`*name*	Record or structure selection operator
`^` `*`	`*`*var* or *var*`^`	Pointer dereference operator
`[`*n*`,...]` or `[`*n*`]...`	`var[`*n*`]`	Array access operator

The rest of this section illustrates the use of all three of these operators.

The following example shows how to display a structure referred by a handle. Suppose your target program contains a variable `menu` of the type `MenuInfo`, which is really a handle, that is, a pointer to a pointer. Thus, if you suspend program execution at a point where `menu` is in the current name scope, and then type `menu` and press the Enter key, SADE responds by displaying a line in this format:

```
^^MenuInfo($00078278)
```

The two pointer operators in this line (`^^`) show that `menu` is a handle. To display a handle, you can dereference it twice, as in:

```
menu^^
```

SADE then displays a menu record, which has the following format:

```
RECORD
      menuID: 129;
      menuWidth: 122;
      menuHeight: 264;
      menuProc: ^^"`\r";
      enableFlags: 4097;
      menuData: 'File';
END
```

The following example shows how to reference a field in a record. If you want to see the value of a field in a record, rather than an entire record, you can use the record selection operator (`.`) in this fashion:

```
menu^^.menuWidth
```

SADE then provides the requested information:

```
122
```

Because handles are used so often in Macintosh programming, SADE offers a shorthand notation for referring to handles. When you want to specify a handle of a record member, as in the previous example, you can leave out the dereferencing operators; in other words, the following two references are equivalent. Instead of typing

```
menu^^.menuID      # longer reference
```

You can use the shorter form:

```
menu.menuID        # shorter reference
```

The response from SADE is:

```
129
```

Because SADE allows this shortcut, you cannot use the familiar C style of dereferencing operators with handles unless you change the normal precedence by using parentheses. For example, if you enter this request, SADE returns an error message:

```
**menu.menuID
```

The error message is:

```
### The dereference operator cannot be applied to this type. menuID
```

The proper syntax is:

```
(**menu).menuID
```

SADE then displays the desired information:

```
129
```

Variable references can be as complex as necessary, as long as each type supports the operator applied to it. For instance, the reference `myRecord.myArrayVar[1]` refers to the first element of the array `myArrayVar`, which is an element of the structure `myRecord`. (Note that SADE array variables are 1-based.)

In SADE, you can also use an array-like construction to refer to symbols outside the current stack activation—that is, to symbols that are contained in a recursive procedure. By placing a number *n* inside square brackets between a procedure name and a variable reference, you can specify the *n*th activation of that procedure from the top of the stack. For instance, the following example refers to the variable `menuID` belonging to the third most recursive call to the procedure `DoMenuCommand`:

```
DoMenuCommand[3].menuID
```

In SADE, a variable reference always refers to the variable's value, not to its address. For instance, `temp` refers to the value of the variable temp and `@temp` (or `&temp`, in C) refers to its address. The value of the variable (`temp`) includes information about resources and offsets, while `@temp` is a simple address at a specific point in time. To assign the address of a procedure to a register, you can precede the name with the `@` operator. For example to assign the address of `temp` to the program counter—that is, to set the program execution point at `temp`—you can enter this command:

```
pc := @temp
```

Symbolic information generated by the MPW Pascal compiler does not include information about the use of `WITH` statements, so SADE cannot handle unqualified field references. To gain access to the fields of a record that has been identified with a `WITH` statement, you must include the name of the record variable, just as you would if the `WITH` statement were not present.

SADE does not recognize a variable that is defined as `extern` in a C source files if the variable's defining file was not compiled with the `-sym on` (or `-sym full`) option.

Predefined SADE variables

A number of predefined SADE variables provide access to state information, such as the exception number most recently encountered, the current date, and the process ID for the current target program. You can reference any of these variables by simply entering its name. For example, to obtain the current date, you can enter the `Date` command and SADE displays the date as shown:

```
Date
'03-May-92'
```

Many predefined SADE variables are read-only and cannot be assigned values. Table 3-1 lists the predefined SADE variables.

■ **Table 3-1** SADE variables

Variable name	Description	Read-only or Read/Write
`ActiveWindow`	A string containing the name of the front most (active) SADE window.	Read-only
`Arg[` *n* `]`	The *n*th parameter of the current SADE procedure. `Arg` is used like an array variable to access the parameters of the current SADE procedure numerically rather than by name. Note that `Arg [` *n* `]` arrays, like other SADE variable arrays, are 1-based.	Read/write
`BreakIfNoSource`	Determines how SADE handles code that do not contain symbolic information—for example, out-of-line arithmetic library routines such as `ULMODT`. If the value of `BreakIfNoSource` is 0 (`FALSE`), which is the default setting, a Step Into command does not break if no symbolic information is available at the program counter; Step Into continues stepping until symbolic information is available. If the value of `BreakIfNoSource` is 1 (`TRUE`), SADE breaks whenever no source information is available at the program counter.	Read/write
`Date`	A string containing the current date in the form *dd-mm-yy*.	Read-only

(continued)

■ **Table 3-1** SADE variables (Continued)

Variable name	Description	Read-only or Read/Write
`DisAsmFormat`	A string used to format the output of the `Disasm` command. Each line of the output is divided into three fields: address, hexadecimal representation, and assembly code. You control the presence, order, and format of the three fields with the `DisAsmFormat` flags. The order of the flags you specify is the order in which the fields will appear in the `Disasm` output, except for the 0 flag; the initial order is `OAXC`. You also determine whether to display the source statement numbers that correlate to the assembly instructions. `o,O` Display source statement numbers (case is not significant, nor is the order in which this flag appears). `a, A` Display the address (case is not significant). `x, X` Display the hexadecimal code representation (case is not significant) `c` Truncate the assembly code if necessary to uniform length `C` Show entire assembly code no matter how long. `$` Generate a `$` character before the offset and/or address field values (allowed only before `A` flag). Blanks and tabs are ignored in the string. You may not use a flag more than once and must specify at least one of the two flags `x`/`X` or `c`/`C`. If you specify the assembly code field last, `c` has the same meaning as `C`—the entire assembly code field is displayed. If you specify that the assembly code field appear before one of the other fields, you have the option of truncating it to a uniform length (`c`) or showing it completely (`C`).	Read/write

(continued)

■ **Table 3-1** SADE variables (Continued)

Variable name	Description	Read-only or Read/Write
`Exception`	The exception number of the most recently encountered exception. Possible values include the standard system error IDs (listed in the MPW file SysErr.a), error IDs that you define, as well as the following special SADE exception numbers: 50 A-trap break 51 Instruction trace (step) 55 User interrupt (SADEKey pressed) 58 Address break 63 Nonfatal internal error 64 Fatal internal error	Read-only
`Inf`	Always equal to a SANE infinity.	Read-only
`NArgs`	The number of actual parameters specified for the current SADE procedure or function. NArgs is undefined when no procedure is in use.	Read-only
`ProcessId`	The process identifier for the current target program. It identifies the process that was suspended when SADE was entered. ProcessID is updated when the Target command is executed.	Read-only
`TargetWindow`	A string containing the pathname of the file open just behind the current window.	Read-only
`WorksheetWindow`	A string containing the pathname of the SADE Worksheet window.	Read-only

System symbols

Macintosh system symbols include Toolbox traps, CPU registers, and low-memory global variables. SADE provides a special character, Δ (Option-J), that you can use to identify a symbol as a system symbol.

Using system symbols to set breakpoints

You can use system symbols to set breakpoints on Toolbox and operating system traps. One way to set a break on a trap is to precede the trap name with an underscore character (_). For example, this command sets a breakpoint on the Toolbox trap GetMHandle:

```
Break _GetMHandle
```

To see whether a breakpoint has been set on a trap, you can execute the `List Break` command:

```
List Break
```

SADE responds with:

```
Trap _GetMHandle (†$A949)
```

This message lists the name of the GetMHandle trap, followed by the trap's number. The trap number is enclosed in parentheses and is preceded by the character † (Option-T). In SADE, the character † is a **trap operator** that is used to identify trap numbers.

If you know a trap's number, you can set a break on a trap by specifying it :

```
Break †$A949
```

Register names

Register names are system symbols.

■ **Table 3-2** System registers

Name	Use
D0...D7	Data registers
A0...A7	Address registers
CCR	Condition code register
SR	Status register
SP	Stack pointer
PC	Program counter
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register
FP0...FP7	Floating-point data registers

SADE does not recognize the MMU registers.

Expressions

Expressions are composed of either a single term or an arithmetic combination of terms. A term is either a named symbol, a numeric constant, a string, or a SADE function call. Terms are combined by arithmetic, logical, shift, and relational operators. String terms may be combined only with relational operators or string functions.

Numeric constants

Numeric constants take the form of decimal, hexadecimal, binary, and floating-point numbers. You can use the `Printf` command to convert a numeric constant and display it in a different format.

In SADE, decimal, hexadecimal, and binary numbers are used as follows:

- Decimal numbers are formed as a string of decimal digits (0–9). SADE treats decimal values as 32-bit (signed long word) quantities. It treats decimal values that exceed 32 bits as floating-point values. To enter an unsigned value, you must coerce the right side:
  `x := UnsignedLong(`*value*`)`
- Hexadecimal numbers are specified by a dollar sign (`$`) followed by a sequence of hexadecimal digits (0–9, A–F, or a–f). SADE treats hexadecimal numbers as 32-bit quantities and "left-pads" them with zeros if necessary; in other words, `$FF` is treated as `$000000FF`. For readability, you can use periods to separate digits without changing their values. For example, the command `Break †$A.9.4.9` is the same as the command `Break †$A949`
- Binary numbers are identified by a percent sign (%) followed by a sequence of binary digits (0–1). SADE treats binary numbers as 32-bit quantities and left-pads them with zeros if necessary. For readability, you can use periods to separate digits without changing their values. For example, `%1.0.0.1` is interpreted the same as `%1001`.
- Floating-point numbers are identified with a decimal point or exponent as described in the *Apple Numerics Manual,* second edition (SANE®). Within SADE, floating-point numbers are represented as SANE extended numbers.

SADE treats hexadecimal and binary numbers that are longer than 32 bits as strings; this is useful for arbitrary assignment of values. If you use these strings in an expression, SADE treats them as SANE extended numbers.

Strings

In SADE, a string is a sequence of one or more ASCII characters (including blanks) enclosed in single (') or double (") quotation marks. Strings are limited to a length of 254 characters. Many SADE commands, such as `Directory`, `Target`, `Open`, `Close`, and `Printf` take a string as a parameter, as do many of the built-in SADE functions, such as `Concat`, `Confirm`, and `Copy`.

```
Directory "SADE 1.3:CExamples"
Target 'sample'
Open 'sample.c'
```

Normally, when you pass a string value to SADE, SADE strips the quotation marks and returns the characters. However, the backslash character (\) is an escape character. The letters `n`, `t`, and `f` also have special meanings when preceded with a backslash character. Also, when you precede a series of one to three digits with a backslash character ("*ddd*") or with a dollar sign and one or two hexadecimal digits ("\$*hh*"), SADE interprets the digits as their ASCII character equivalents.

In strings enclosed in double quotation marks, SADE interprets a pair of backslashes `("\\")` as a single backslash, and interprets a single backslash `("\")` as a single quotation mark. Note, however, that this is true only of strings enclosed in double quotation marks. If you enclose a backslash in single quotation marks, SADE simply interprets it as a backslash character.

If you precede the letters `n`, `t`, and `f` or their ASCII equivalents with a backslash, SADE interprets them as follows:

`\n` or `\$0D`	newline
`\t` or `\$09`	tab
`\f` or `\$0C`	formfeed

One use for the backslash escape character is to pass ASCII control codes to format character strings. For example, if `myVar` = 123 and `yourVar` = 55, you can use this command to insert tabs and a newline into a string formatted with the `Printf` command

```
Printf "Value of myVar:\t\t %#X\nValue of yourVar:\t %#X",myVar,yourVar
```

SADE responds with this output:

```
Value of myVar:     $7B
Value of yourVar:   $37
```

You can delimit a string with double quotation marks and use single quotation marks as literals inside the string, or vice versa. For example, the string

```
"Use 'Format' instead"
```

is interpreted by SADE as:

```
Use 'Format' instead
```

You can use string constants in arithmetic expressions, but any string you use in this fashion can contain at most four characters. SADE treats such strings as right-justified 32-bit signed values. SADE assigns each character in such a string its ASCII value and represents the overall value of the string as the concatenation of the values of its elements. For instance, the string `'my'` has the value $6D79 because the ASCII equivalent of `m` is $6D and the ASCII equivalent of `y` is $79).

- *Note:* Each command line in a file is limited to a maximum of 254 characters. Furthermore, a string of 254 characters is too long to be executed from a file, because the quotation marks used as delimiters are counted as part of the line length.

SADE treats C strings, which are null-terminated arrays of characters, as arrays of unsigned bytes. It does, however, attempt to display arrays of 1-byte values, as well as pointers and handles to such values, as either C or Pascal strings.

Built-in functions

SADE provides a set of built-in functions which, unlike commands, can be used in expressions. For example, you can take the output generated by a function and assign it to a variable:

```
myVar := SourceToAddr(targetWindow)
```

The SADE built-in functions are described in Part II, "Command Reference."

Operator precedence

This section describes the operators used to form expressions in SADE. Table 3-3 lists the SADE operators in precedence from highest to lowest. Groupings within the table show operators of the same precedence.

■ **Table 3-3** Operator precedence

Operator	Meaning
( )	Used to group expressions (includes casting and argument lists)
->	Qualifier by pointer
.	Qualifier
++	Increment
--	Decrement
†	Trap
@	Address of
^	Pointer to
¬, NOT, !	Logical NOT
~	Bitwise one's complement
*	Pointer to
+	Unary positive
-	Unary negative
&	Address of
*	Multiplication
/, DIV, ÷	Division
//, MOD	Remainder

(continued)

■ **Table 3-3** Operator precedence (Continued)

Operator	Meaning
+	Addition
-	Subtraction
>>	Shift right
<<	Shift left
=, ==	Equal
<>, ≠, !=	Not equal
<	Less than
>	Greater than
<=, ≤	Less than or equal
>=, ≥	Greater than or equal
&, AND	Bitwise AND
&&	Logical AND
|, OR	Bitwise OR
||	Logical OR
XOR EOR	Bitwise exclusive OR
?:	Condition
:=	Size-compatible assignment
<-	Arbitrary assignment
<*op*>=	Assignment with operation (where <*op*>= is one of these: += -= *= /= DIV= ÷= //= MOD= <<= >>=)
..	Range

◆ *Note:* You can use the range operator (. .) only once in an expression.

Expression operand basic types

SADE provides the basic types shown in Table 3-4. SADE also recognizes the types that you define in the target program.

■ **Table 3-4** SADE basic types

Type	Definition
Boolean	A 1-byte Pascal Boolean value
UnsignedByte, UnsignedChar	A byte in the value range 0-255
Byte	A byte in the value range –128-127
CChar	A byte in the value range 0-255
PChar, PascalChar	A word in the range 0-255 (Pascal `Char`)
UnsignedWord, UnsignedShort	A word in the range 0-65,535
Word, Short, Integer	A word in the range –32,768-32,767
UnsignedLong, UnsignedInt, Unsigned, UnsignedLongInt	A long word in the range 0-4,294,967,295
Long, Int, LongInt, SignedLongInt	A long word in the range –2,147,483,648-2,147,483,647
Single, Float, Real	An IEEE floating-point single-precision value (4 bytes)
Double	An IEEE floating-point double-precision value (8 bytes)
Extended	An IEEE floating-point extended-precision value (10 bytes)
Extended12	An IEEE floating-point extended-precision value (12 bytes)
Comp[utational]	A SANE signed 8-byte integer
CString	Up to 254 characters terminated by a null byte
PString, String, Str255	A length byte followed by up to 254 characters

In addition to the definitions listed in this table, you can define pointer types with the pointer operators (see "Type Coercion" later in this chapter).

▲ **Warning** The SADE type `Byte` is frequently overridden by the Pascal type `Byte` (which is two bytes long). You can use `SizeOf(`*byte*`)` to tell if this overriding occurred, or you can use the SADE type `UnsignedByte` instead. ▲

Expression evaluation

This section describes how SADE evaluates a single-term or multiterm expression. A single-term expression is represented by a single symbol and takes on the value represented by the symbol (the value associated with the name, constant, or string). If a symbol represents an array or record structure, the "value" of the expression is the entire array or record structure.

Integers and floating-point numbers

A multiterm expression consists of two or more operands. SADE uses different rules to reduce a multi term expression to a single value, depending upon whether the operands are integers or floating-point numbers.

SADE follows this set of rules when evaluating multiterm integer expressions:

- Each signed integer operand is converted to a 32-bit signed value.
- Each unsigned integer operand is converted to a 32-bit unsigned value.
- Each floating-point and computational value is converted to a 10-byte extended value.
- When a binary operator combines two integer operands, both operands are treated as unsigned if either is unsigned. The result is then treated as a 32-bit unsigned value.
- If both integer operands combined with a binary operator are signed, then the result is a signed 32-bit value.
- If either operand is a floating-point value, then the other operand is converted to a floating-point extended value and the result is extended.
- Integer division always yields an integer result; any fractional portion of the result is dropped.
- Integer division by 0 yields 0 as the result.

SADE follows this set of rules when evaluating multiterm floating-point expressions:

- Each floating-point and computational value is converted to a 10-byte extended value.
- Floating-point division by 0 yields infinity, except for 0 divided by 0, which yields an NaN value.

Precedence

When SADE evaluates an expression, operations are performed from left to right following the precedence indicated in Table 3-3, except for assignment operations, which are performed from right to left.

- A parenthesized sub-expression is reduced to a single value. The resulting value is then used to compute the final value of the expression.
- When parenthesized sub-expressions are nested, the innermost sub-expression is evaluated first.

Operators

These logical operators evaluate to the value 1 (`TRUE`), or the value 0 (`FALSE`).

- NOT, ¬, !
- =, ==
- <>, ≠, !=
- >
- <
- <=, ≤
- >=, ≥
- &&
- ||

The << operator shifts zeros in; bits shifted out are lost. The >> operator maintains the sign of the expression—0 if positive, 1 if negative.

C programmers should note that pointer arithmetic using the +, –, and <op>= operators works as it does in Pascal. Pointer arithmetic using the ++ and –– operators works as you would expect in C.

The assignment (:=,<–,<op>=), pointer (^,*), trap (†), and address (@,&) operators are special operators with meanings unique to SADE. They are discussed separately in the sections that follow.

The assignment operator

SADE treats the assignment operator as a binary operator, so you can embed an assignment operator in a more complex expression to capture intermediate results. The assignment operator is the only operator that is evaluated from right to left. Thus an expression of the form

$a := b := c := d$

is evaluated as if it had been written like this:

$a := (b := (c := d))$

The left operand of an assignment must be a variable reference. For an integer reference, the right operand is saved in the specified variable. For a floating-point assignment, the right operand is converted to an extended value before the assignment, if necessary. For string, record, or array assignments, the left variable must be compatible with the right operand, and no other operators can be combined with the assignment.

The above rules also hold for the <*op*>= forms of operators.

Compatibility between operands in SADE is defined as it is in Pascal for real and integer values. For structured data, compatible operands are defined as having the same aggregate size. A second operator is provided for *arbitrary* assignment, namely <–. Using this assignment operator, you may assign any type to any other type, regardless of size. For arbitrary assignments, the size of the right operand is used to determine the number of bytes to move. You can use this operator, for example, to patch memory.

- *Note:* These restrictions, as well as the discussion of the <– operator, do not apply to SADE variable assignment because SADE variables are dynamically typed; they automatically take on the type of the value assigned to them.

The pointer operator

There are two forms of pointer operators, one following Pascal conventions and the other following C conventions. When you use the Pascal pointer symbol (^) as an operator, it must follow an expression term—for example `myWindowPtr^`. In C, the pointer symbol (*) precedes the variable reference—for example `*myWindowPtr`. If a program variable has more levels of indirection, you can use pointer operators to dereference as many layers as necessary.

To refer to record structures referenced by pointers, you can use either the standard pointer operator with the qualifier operator (`.`) or the qualifier-by-pointer operator (`->`). When you use these operators, it is safer to use the Pascal convention (`^`) with the standard qualifier, not the C-style pointer qualifier, because precedence rules make it easier to get a correct reference when you use the Pascal convention. In a Pascal-style reference, SADE first evaluates the record qualifier, and then evaluates the Pascal pointers at which point it displays the record. In a C-style reference, because the pointer operator has a lower precedence, SADE evaluates the record qualifier, performs the special handle processing, and then evaluates the C-style pointer. At this point SADE has already typed the variable as a handle, so it returns an error when requested to dereference it twice more.

It is especially important to use the Pascal-style convention for accessing records when you are dealing with handles (pointers to pointers). For example, this kind of reference to a handle always works:

```
menu^^.menuID
```

If you specify a field of a record structure pointed to by a handle, SADE automatically does the dereferencing for you. For example:

```
menu.menuID
```

Other conventions don't work or cause problems:

```
**menu.menuID      # C convention doesn't work without parentheses
(**menu).menuID    # works but takes thought
*menu->menuID      # C convention does not work
(*menu)->menuID    # works but takes thought
menu.menuID        # works because of special handle processing
```

When a term being referred to by a pointer operator is a variable reference, the pointer operator indicates an indirect reference through the variable, and the type of the term is determined by the type associated with the pointer variable reference. When the term is a subexpression, the pointer operator indicates an indirect reference through the address represented by that subexpression. In this case, the type is assumed to be a pointer to a long integer. You can use type coercion (described later in this chapter) to treat the reference as some other type.

The address operator

You can generate a pointer to a variable (an address) with the address operators (`@`) and (`&`). Address operators are unary operators taking a variable or procedure reference as the operand. For example, this statement assigns the address of `DoMenuCommand` to the program counter:

```
pc := @DoMenuCommand
```

The type of the value is considered to be a pointer to the type of the variable. You cannot apply the address operators to SADE variables or to the addresses of variables stored in registers.

The trap operator

You can create an expression whose value is a trap by using the trap operator (`†`). The trap operator is a unary operator taking an expression element or a parenthesized expression as the operand. Such trap expressions are used with breakpoint commands to distinguish trap breakpoints from address breakpoints.

Type coercion

You can use the names of known types in a function-like notation to perform **type coercions** on, or change the types of, expressions. For example, SADE treats decimal numbers as signed long words. To assign an unsigned long decimal value to a variable, you must coerce the right side of the expression:

```
myVar := UnsignedLong(-12345)
```

You can specify any predefined type names or any types defined in your program. SADE coerces the type of an object as long as there is a reasonable way to interpret and perform the coercion.

Additionally, you can precede the type specification with the pointer operator (`^`) to indicate coercion to a pointer of the specified type. This is an extension of the Pascal notation that allows type declarations such as `^integer`. (In fact, you can precede a type name by up to three pointer operators to construct new pointer types.)

Table 3-5 shows how the type coercion mechanism works in conjunction with indirect memory references.

■ **Table 3-5** An example of type coercion

Pascal	C	Result
`comp(10)`	`(comp)10`	Converts the number 10 to the computational type
`comp(10^)`	`(comp)(*10)`	Converts the long at location 10 to the computational type
`^comp(10)`	`(comp*)10`	Identifies 10 as a pointer to a computational type
`^comp(10)^`	`*(comp*)10`	Returns the computational type at location 10

Ranges

Certain SADE commands take ranges as arguments. You can express ranges of addresses or values with a pair of expressions (the low and high ends of the range) separated by the range operator (`..`). The syntax is as follows:

expr `..` *expr*

You cannot use a floating-point value on either side of the range operator to designate a range. If one end of the range expression is a trap number, both must be trap numbers, as in this example:

```
†$A000..†$AFFF
```

Chapter 4 **C Tutorial Examples**

The tutorial in Chapter 1 gives examples of the most basic debugging concepts using only the menu commands. The purpose of the tutorials in this chapter is threefold:

- to provide examples of how to use the SADE language
- to give additional examples of menu usage
- to demonstrate some debugging strategies

The sample programs described in this chapter are written in C. See the next chapter for Pascal exercises.

Debugging sample applications using SADE

The applications used for the debugging exercises in this chapter are in the CExamples folder that is inside the SADE Tutorials folder. The source code listings in the sections "C Tutorial 1" and "C Tutorial 3" are titled Sample1.c and Sample2.c. The applications they generate are named Sample1 and Sample2. These are variations of the Sample program in the MPW CExamples folder. The Sample.c application draws a traffic light on the screen and switches the color of the light from red to green and back (or, on black-and-white monitors, between white and black). It demonstrates how to initialize some of the most commonly used Toolbox managers and how to write a program that operates successfully under MultiFinder, handles desk accessories, and create, grow, and zoom windows.

However, in Sample1.c and Sample2.c, bugs have been intentionally introduced. The bugs in the programs are caused by common programming errors.

The application debugged in the section "C Tutorial 2" is titled CStuff. The CStuff application contains a collection of variable and function definitions that demonstrate how to reference non-local code and variables. The CStuff application is compiled from three source files: Stuff.c, MoreStuff.c, and MoreStuff.h.

Remember these points about debugging the sample applications in this chapter:

- These applications have been successfully compiled and linked. SADE cannot debug an application that won't compile and link.
- The applications were compiled and linked using the `-sym on` option. This option generates a symbol file that SADE requires to debug an application effectively. Whenever you plan to use SADE to debug a program, you should compile and link the program with `-sym on`.

C Tutorial 1

This tutorial demonstrates how to perform basic debugging operations using the SADE command language. It shows you how to:

- set a directory and target the application to debug using SADE commands instead of menus
- set breakpoints and tracepoints using SADE commands
- step through the application

Setting the directory and targeting a program

Before you can target Sample1, you must set the working directory to the CExamples folder that is inside the SADE Tutorials folder. If SADE is not set to the correct directory, SADE cannot automatically open the necessary source files during a debugging session.

If you select the Target menu command from the File menu, SADE sets the current directory to that of the selected program. If you debug the same program repeatedly, you may actually save time by writing and reusing two SADE commands, Directory and Target, which perform the same operations. Type the following lines in your worksheet and hit Enter.

```
Directory 'hdName:SADE:SADE Tutorials:CExamples:C tutorial 1:'
Target 'Sample1' using 'Sample1.sym'
```

Replace *hdName* with the name of the hard disk that contains the SADE application.

◆ *Note:* For precision's sake, this example explicitly identifies the symbol file name as well as the name of the program to debug. As a rule, you can leave out the symbol file name because SADE automatically looks for the default name *programName*`.sym`. You are required to specify the symbol file name (with the `using` option) only if you have given the symbol file a different name; if the program and its symbol file are in different directories; or in special cases, such as debugging an MPW tool, where the target application is MPW and the symbol file is that of the tool.

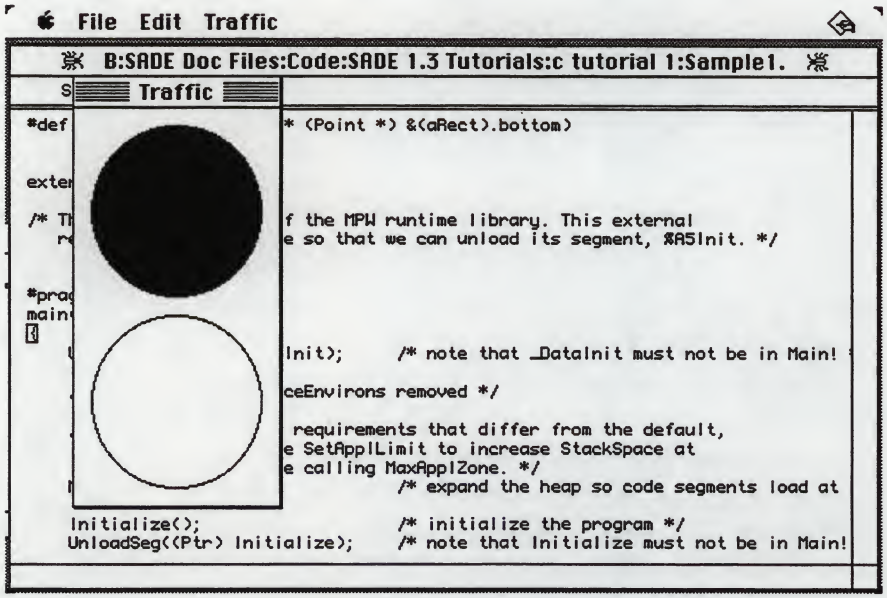
- *Note:* In this tutorial, the Sample1 application has only one source file. Most applications have multiple source files. As long as all the source files reside in one directory, specifying a default directory lets SADE know where they are. However, if you store the source files for an application in multiple directories, you can use the `SourcePath` command to specify all the directories that SADE searches for the source files. (For more information on `SourcePath`, see Part II, "Command Reference.")

In addition to identifying a program to debug, the `Target` command also launches it, stopping at the first statement of `main()`. SADE highlights that statement. To continue execution of the program, you can do one of the following:

- select Go from the SourceCmds menu
- use the Go command's keyboard equivalent, Command-P
- type `Go` in the SADE Worksheet window and hit Enter

The Sample1 application and its menu bar appear as shown in Figure 4-1.

Figure 4-1 Sample1 application

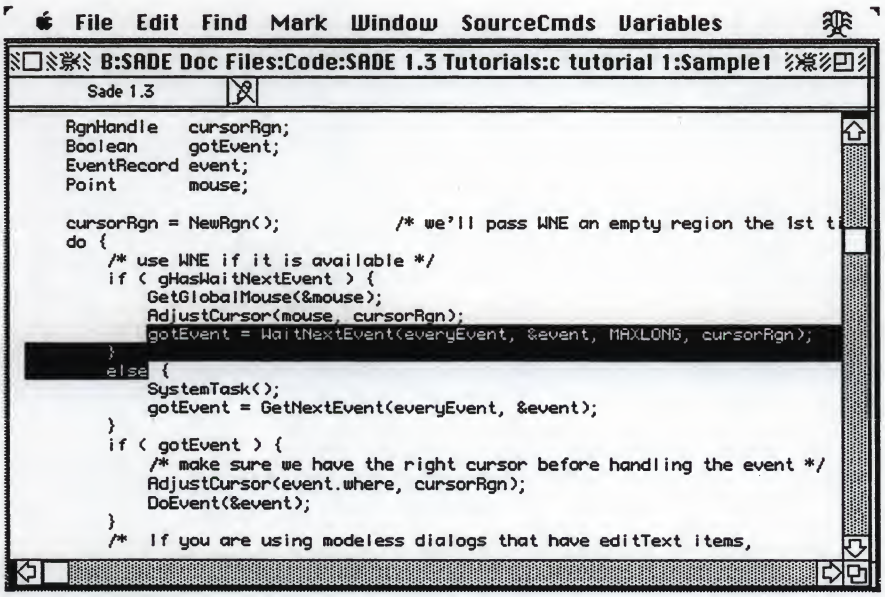


Sample1 has one simple function: to switch the traffic light between red and green (if you have a black-and-white monitor, the top circle is darkened when red, and the bottom circle is darkened when green). In versions of the program that do not contain a bug, you can switch the traffic light between red and green by clicking either circle or by choosing the Green Light or Red Light command from the Traffic menu. If you experiment with Sample1, you will discover that you can switch the lights by clicking either circle. The Green Light command works correctly as well. But you will find that the Red Light command in the Traffic menu turns on the green (bottom) light instead of the red one.

Looking at the source file and making a guess

Now that you know there is a bug in the Sample1.c program, how do you go about finding it? When this tutorial was first demonstrated in Chapter 1, it did not go through the logical steps to find the bug. This tutorial, however, does go through these steps. But first, type the SADEKey combination (Command-Option–keypad period) to suspend the execution of Sample1 and bring its source file window to the front. The SADEKey combination suspends an application the next time the application calls a `WaitNextEvent`, `GetNextEvent`, or `EventAvail` routine. When SADE opens the source file, it highlights the statement that called the event trap—in this case, `WaitNextEvent`, as shown in Figure 4-2.

■ **Figure 4-2** Sample1.c source file

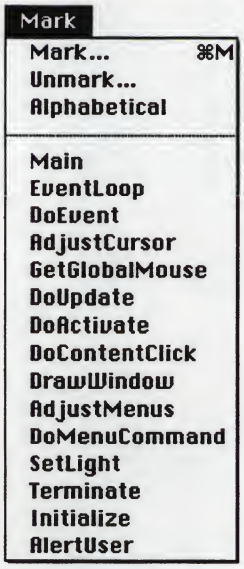


To begin tracking down a bug, you have to guess its cause based on what you know about the program. Two significant facts about this application provide a clue to the source of the bug:

- Clicking either circle changes the light correctly.
- Selecting the Red Light menu command does not work.

A solid guess is that something is wrong in a routine that handles menus. Therefore, it might be a good idea to search for routines that have the word menu in their names. You could use the menu command Find (under the Find menu) to search for the word menu. However, SADE recognizes markers placed in a file by the MPW Shell. For your convenience, all of the routines in the source file Sample1 have been marked, and you can see a list of the current markers in the file by pulling down the Mark menu, as shown in Figure 4-3.

- **Figure 4-3** The Mark menu



As you can see, there are two routines—`AdjustMenus` and `DoMenuCommand`—whose names suggest that they deal with menus. Since the bug is associated with a menu command, choose `DoMenuCommand`, the more likely candidate of the two, from the list in the Mark menu.

Setting a breakpoint

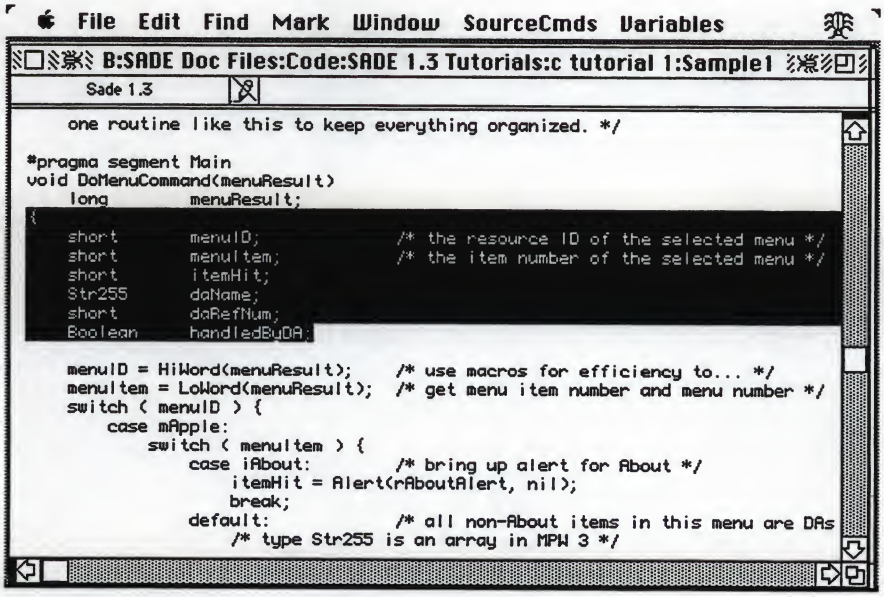
The strategy for finding the bug in Sample1 is first to verify that `DoMenuCommand` handles the Red Light and Green Light menu commands; if it does, then you need to determine which statements in `DoMenuCommand` explicitly handle those menu commands. If `DoMenuCommand` appears to have nothing to do with these menu commands, then the same test can be applied to `AdjustMenus`.

To test `DoMenuCommand`, you can put a breakpoint in `DoMenuCommand`, run Sample1, and see if the application halts when you choose a menu command.

To set the breakpoint, click anywhere on the first statement of the `DoMenuCommand` routine in your source file, and select Break from the SourceCmds menu (or use the keyboard equivalent Command-B). Note that the insertion point must be on or after the brace, not in the name of the routine—or SADE cannot set the breakpoint.

Now resume program execution by choosing the Go command or pressing Command-P. When Sample1 resumes execution, you can test the hypothesis that `DoMenuCommand` handles mouse clicks on menus. Just click on any menu title without selecting a menu command. SADE again suspends execution of Sample1, brings the Sample1.c source file to the front, and highlights the first statement of `DoMenuCommand`, as shown in Figure 4-4.

■ **Figure 4-4** Stopping on a breakpoint



Although you have confirmed that `DoMenuCommand` handles menus, you still don't know if or how it handles the Red Light command. To test this, resume program execution again, but this time choose Red Light from the Traffic menu. Again SADE interrupts the program at the first statement in DoMenuCommand, so you know that some statement in `DoMenuCommand` handles the Red Light command.

Stepping through the application and setting a tracepoint

Now that SADE has control of the application, you can see the statements that Sample1 executes to process the Red Light command by stepping through `DoMenuCommand` one statement at a time. Each time you choose Step from the SourceCmds menu (or use the keyboard equivalent, Command-L), SADE highlights the next statement to execute. If you step until all the statements in `DoMenuCommand` have been executed, this sequence of statements is highlighted on the screen:

```
(Step)
        menuID = HiWord(menuResult); /* use macros for efficiency to..*/
(Step)
        menuItem = LoWord(menuResult);          /* get menu item num and menu num*/
(Step)
        switch ( menuID ) {
(Step)
                case mLight:
                        switch ( menuItem ) {
(Step)
                                case iStop:
                                        SetLight(FrontWindow(), true);
(Step)
                                case iGo:
(Step)
                                        SetLight(FrontWindow(), FALSE);
(Step)
                                        break;
                        }
(Step)
                        break;
        }
        HiliteMenu(0);          /* unhighlight what MenuSelect (or MenuKey) */
                                        /* highlighted */
(Step)
} /*DoMenuCommand*/
```

Resume program execution again, but this time select the Green Light command. SADE interrupts program operation at the first statement of `DoMenuCommand` . If you step through `DoMenuCommand` again, you'll see the highlighting move through the following sequence of statements:

```
(Step)
        menuID = HiWord(menuResult); /* use macros for efficiency to..*/
(Step)
        menuItem = LoWord(menuResult);          /* get menu item num and menu num*/
(Step)
        switch ( menuID ) {
(Step)
                case mLight:
                        switch ( menuItem ) {
(Step)
                                        SetLight(FrontWindow(), false);
(Step)
                                        break;
                        }
(Step)
                        break;
        }
        HiliteMenu(0);              /* unhighlight what MenuSelect (or MenuKey) */
                                                /* hilited */
(Step)
} /*DoMenuCommand*/
```

It appears that choosing Red Light calls the `SetLight` routine twice, while choosing the Green Light command calls it once. You can verify this easily by setting a **tracepoint** at `SetLight`. When you put a tracepoint on a statement, SADE informs you each time the target program executes that statement. To trace `SetLight`, you should first remove the breakpoint so the program runs without interruption; then issue the `Trace` and `Go` commands. In the Worksheet, type these commands and press Enter:

```
Unbreak All
Trace SetLight
Go
```

When the application runs, choose the Red Light command. During the execution of the command two tracepoints appear in the SADE Worksheet:

```
TRACE:      SetLight.(0)
TRACE:      SetLight.(0)
```

- *Note:* SADE uses indexes relative to the start of a routine to refer to statements within a routine—and the indexes are 0-based. Since the tracepoint is on the first statement of `SetLight`, the index for it is 0 (`SetLight.(0)` or simply `SetLight`). A routine name without an index indicates the first statement of the routine.

When you select Green Light, a single trace appears in the Worksheet:

```
TRACE:      SetLight.(0)
```

If Sample1 calls SetLight twice when you choose the Red Light menu command, perhaps it changes the light to red the first time and changes it to back to green the second time. Notice that the `DoMenuCommand` procedure has a switch statement whose case statement (case `mLight`) determines the action to take based on which menu the user selects in the Sample1 application. To test the hypothesis that the Red Light command changes the stop light to red and then back to green, you may want to watch the changing value of the variable `gStopped`, which tells whether the stop light is currently on stop or go. To do this, interrupt the application with the SADEKey and remove the tracepoint:

```
Untrace SetLight
```

Now highlight `gStopped` in the source file, or type it in the Worksheet and highlight it, and then choose Add Watch Variable from the Variables menu. SADE opens a window that displays the current value of `gStopped`. Now there are three windows open; if you wish, you can choose Tile Windows from the Window menu to organize the display better.

To narrow the focus, place a break on the following statement in the `DoMenuCommand` procedure:

```
case mLight:
```

Resume program operation (Command-P) and make sure that the light is green; if it is red, click one of the circles to change the light to green. Choose Red Light from the Traffic menu. The program then breaks on the `case mLight` statement. Note that the value of `gStopped` is 0. Advance the program one step by typing Command-L, and the following code is highlighted:

```
case iStop:
      SetLight(FrontWindow(), true);
```

Step into `SetLight` (Command-Option-L) and then step three more times. After the third step, note that the value of `gStopped` has changed to 1, which means that the light is now red. Choose Step Out from the SourceCmds menu to step out of `SetLight`, and the following code is highlighted:

```
case iGo:
```

Step twice more (the program calls `SetLight` again) and you'll see that `gStopped` is 0, which means that the light has changed back to green.

Clearly, the `case mLight` statement is working properly, calling `SetLight` and changing the light to red. The problem is that the program immediately calls `SetLight` again and changes the light back to green. The solution is to put a break between `iStop`, which determines what happens when the Red Light menu command is chosen, and `iGo`, which determines what happens when the Green Light menu command is chosen.

To finish the debugging session, quit Sample1 or execute the SADE `Kill` command. Either of these actions kills Sample1 and ends the debugging session.

To quit SADE, choose Quit from the SADE File menu or issue the scriptable `Quit` command:

```
Quit
```

C Tutorial 2

This tutorial demonstrates how to work with program symbols that may not be directly accessible because they are outside the current scope.

Before you start the tutorial, you may want to review the material on current scope and qualified variable references presented in Chapter 3, "Language Overview."

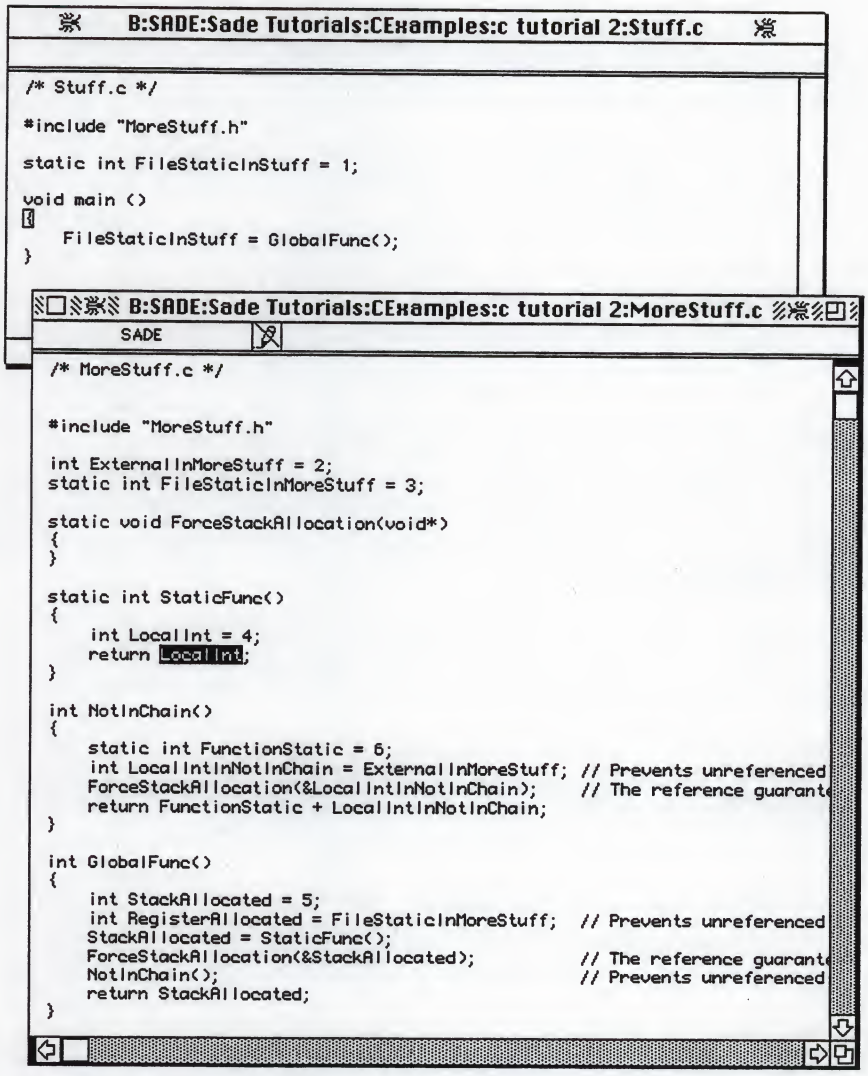
The application used is CStuff. It contains a collection of variable and function definitions that are used to demonstrate

- how SADE handles global variables
- when and how to qualify variables that are local to a file or a function so that they can be accessed from other files or functions.
- how SADE handles variables that are stored in registers

The CStuff application, its sources (Stuff.c and MoreStuff.c) and header file are in the SADE Tutorials folder.

Figure 4-5 shows the source code for Stuff.c and MoreStuff.c.

■ **Figure 4-5** Source code for C Tutorial 2



B:SADE:Sade Tutorials:CExamples:c tutorial 2:Stuff.c

```
/* Stuff.c */

#include "MoreStuff.h"

static int FileStaticInStuff = 1;

void main ()
{
    FileStaticInStuff = GlobalFunc();
}
```

B:SADE:Sade Tutorials:CExamples:c tutorial 2:MoreStuff.c

SADE

```
/* MoreStuff.c */


#include "MoreStuff.h"

int ExternalInMoreStuff = 2;
static int FileStaticInMoreStuff = 3;

static void ForceStackAllocation(void*)
{
}

static int StaticFunc()
{
    int LocalInt = 4;
    return LocalInt;
}

int NotInChain()
{
    static int FunctionStatic = 6;
    int LocalIntInNotInChain = ExternalInMoreStuff; // Prevents unreferenced
    ForceStackAllocation(&LocalIntInNotInChain);    // The reference guarant
    return FunctionStatic + LocalIntInNotInChain;
}

int GlobalFunc()
{
    int StackAllocated = 5;
    int RegisterAllocated = FileStaticInMoreStuff;  // Prevents unreferenced
    StackAllocated = StaticFunc();
    ForceStackAllocation(&StackAllocated);          // The reference guarant
    NotInChain();                                   // Prevents unreferenced
    return StackAllocated;
}
```

You can make the exercise easier by using the file CTutorialScript included in this folder, which contains all the commands executed in the tutorial. To use this file, simply open it, find the SADE command or series of commands that you want to execute, and copy the text you have selected to the Worksheet. You can then execute your command or series of commands from the Worksheet.

Function/code references

In terms of scoping, there are only three types of functions:

- functions which are defined in the current file
- global functions which are defined in another file
- static functions which are defined in another file

As long as all the files have been compiled with symbols, you can make references to all of these cases, some requiring more specification than others.

◆ *Note:* Actually, C++ programs have additional scoping rules related to ordinary member functions and static member functions. The subtleties of these rules are explained in Chapter 6, "Special Debugging Cases."

To start the tutorial, target the program CStuff.

```
Directory 'hdName:SADE:SADE Tutorials:CExamples:C Tutorial 2'
Target 'CStuff'
```

where *hdName* is the name of your hard disk.

The program stops in the first line of `main`, the current file at this time is Stuff.c. Since `main` is a function defined in the current file, you can reference it using a simple reference:

```
DisAsm main.(1) 3       # Disassemble 3 instrs starting at main.(1)
```

SADE displays

```
main.(1)
      00808DE6  4EBA FFBC    JSR    GlobalFunc.(0)      ; 00808DA4
      00808DEA  2B40 FE72    MOVE.L D0,-$018E(A5)
main.(2)
      00808DEE  4E5E         UNLK   A6
```

The function `GlobalFunc` is defined in a separate file, but it is defined as a global. You can make a simple reference to it as well:

```
Break GlobalFunc        # Sets break at the start of GlobalFunc
```

The function `StaticFunc` is also defined in a separate file, however, it is local to MoreStuff.c and is therefore outside the current program scope. If you try to reference this function SADE returns an error message:

```
Break StaticFunc        # Try setting break point in StaticFunc
### Could not find "StaticFunc" as a program symbol
```

To be able to set a breakpoint on `StaticFunc` you need to fully qualify its name by including the name of the file which contains the variable, as in the following command:

```
Break \MoreStuff.StaticFunc    # Breaks in StaticFunc
```

Before you proceed, please remove the break points by selecting Unbreak All from the SourceCmds menu.

Variable references

The different circumstances under which you may want to see the value of a variable are:

- the variable is local, that is, defined in the current function. In this case, SADE can display the variable regardless of whether it is stack based (that is, stored in memory) or register based.
- the variable is global. There are many instances of this case, for example you can have a global which is external and visible everywhere in the program. The variable may be static to a file; this requires you to qualify the name to find it.
- the variable is defined as a static inside a function that is not the current function. This function may or may not be in the current call chain. This also requires you to qualify the name to find it, the syntax used is different.
- the variable is an automatic variable in a function other than the current function. The only time you can display an automatic variable belonging to a non-local function is when that function is in the current call chain. Even for functions that are in the call chain, there are two different cases. If the function allocated the variable on the stack, SADE can compute the memory address to look for it and displays the value. If the variable gets allocated in a register, some intervening function might have reused this register. SADE cannot figure out the various possibilities.

This tutorial demonstrates all these cases with the program suspended at a single location inside the file MoreStuff.c. Go to this location by executing the command:

```
Go Til \MoreStuff.StaticFunc.(2)
```

The simplest type of reference is to a local variable; you can simply enter the variable name

```
LocalInt
```

and SADE displays

```
4
```

You can also display the value of any global variable defined in this file (MoreStuff.c) by typing its name. To obtain the value of `ExternalInMoreStuff`, enter its name

```
ExternalInMoreStuff
```

SADE displays

```
2
```

Variables defined as static in this file are visible. To obtain the value of `FileStaticInMoreStuff`, enter its name

```
FileStaticInMoreStuff
```

SADE displays

```
3
```

However, variables defined as static in other files can not be referenced directly:

```
FileStaticInStuff
```

SADE displays

```
### Could not find "FileStaticInStuff" as a program symbol
```

To reference this variable, you need to specify the file the variable is defined in:

```
\Stuff.FileStaticInMoreStuff
```

SADE displays

```
1
```

- *Note:* An interesting point about this example, in terms of the C language, is that the current file has no visible declaration of the variable `FileStaticInStuff` defined elsewhere, and any reference to it would be illegal C.

You can display static variables belonging to other functions. For example, type

```
FunctionStatic
```

press Enter, and SADE displays

```
### Could not find "FileStaticInStuff" as a program symbol
```

You need to specify the function in which the variable is defined (because multiple functions can define the same name), as follows:

```
NotInChain.FunctionStatic
```

SADE displays

```
6
```

Try displaying an automatic variable belonging to another functions. For example,

```
LocalIntInNotInChain
```

SADE displays

```
### Could not find "FileStaticInStuff" as a program symbol
```

Qualify the name specifying which function the variable is defined in

```
NotInChain.LocalIntInNotInChain
```

SADE displays

```
### The variable, "LocalIntInNotInChain", is A6 based and its procedure
is not in the call chain
```

Use the `Stack` command to see the call chain.

```
Stack
Frame At  Owned By                Called By
$00DC4884 main                    %__MAIN+$003C
$00DC487C GlobalFunc              main.(1)
$00DC486C StaticFunc              GlobalFunc.(1)
```

Since `GlobalFunc` is in the call chain, try to display a variable in that function:

```
GlobalFunc.StackAllocated
```

Now SADE responds

```
5
```

However, try to display the variable `RegisterAllocated` in `GlobalFunc`:

```
GlobalFunc.RegisterAllocated
```

SADE prints

```
### The variable, "RegisterAllocated", is in a register and cannot be
referenced except in its own frame
```

This message means that your program must be stopped inside the body of the function `GlobalFunc` for SADE to be able to display the variable `RegisterAllocated`.

- *Note:* Compilers try to allocate local variable to registers as much as possible. However, compilers do not typically allocate a local variable in a register if its address is used anywhere in the body of the function. The sample program uses that assumption to make sure that some of the local variables were allocated in the stack, by way of illustration only, not as a recommended programming technique. In reality compilers can not allocate any structure in registers, so they will be visible. Even for integral variables, the compiler may run out of registers and put it in the stack anyway.

When this debugging session is complete, you can exit SADE by choosing Quit from the File menu or executing the scriptable `Quit` command. SADE will terminate the suspended sample program as a part of its exit behavior.

C Tutorial 3

The Sample2 application, its source, header, and resource files are in the SADE Tutorials folder. To target the program, execute the following commands:

```
Directory 'hdName:SADE:SADE Tutorials:CExamples:C Tutorial 3'
Target 'Sample2'
```

where *hdName* is the name of your hard disk. The `Target` command launches the Sample2 application, opens the source file Sample2.c, and breaks at the first statement of `main`.

When you resume execution of Sample2 (by choosing Go from the SourceCmds menu or executing the `Go` command in the Worksheet), the application crashes and SADE returns a message reporting a bus error or an address error. SADE displays the following information:

```
Bus Error in "Sample2" at
$00010526
          00816F8C  2050              *MOVEA.L       (A0),A0
```

When your program crashes, the most useful information is to see the call chain, or the order in which the routines were called. Then you can examine those routines to look for the bug that crashed the program.

To see the call chain, choose the Show Stack command from the Variables menu. Alternatively, you can execute the scriptable `Stack` command from the worksheet, as follows:

```
Stack
```

Either way, SADE returns the call chain:

```
        Frame At Owned By                  Called By
        $0030FD40 main                      %__MAIN+$003C
        $0030FD38 $00816F8C                 main.(3)
```

This stack, unfortunately, is broken. SADE normally uses the A6 based linkages to display the call chain. In this case the register A6 was being used by the lowest level routines for other purposes. In the SADE Examples script folder, there is a script which demonstrates how to implement the A7 based stack display, as found in the MacsBug sc7 command. Use this script to get a better idea of the call chain. Execute the commands:

```
Execute 'HD:Sade:Sade Example Scripts:sc7'
StackCrawl7
```

The output looks like this:

```
Return addresses on the stack
Stack Addr    Frame Addr    Caller
$0030FD44     $0030FD40     $00303DD8  %__MAIN+$003C
$0030FD3C     $0030FD38     $0030358C  main.(3)
$0030FD06                   $00309C52  Initialize.(29)+$0006
$0030FD02                   $00817F62
```

This stack tells you that the last executing routine in the sample program was `Initialize`, which called some toolbox trap, and the program died. To get a more precise idea of where in the `Initialize` routine the crash occurred, you can trace the Macintosh toolbox traps that Sample2 started up during initialization. However, you must first terminate Sample2 and then retarget it. Otherwise Sample2 will keep crashing because it will keep resuming execution at the point of the crash. To kill and retarget Sample 2, execute these two commands:

```
Kill
Target 'Sample2'
```

You could now instruct SADE to trace all traps and then resume program execution, but you would then have to look at a number of traps that are of no interest. Since you already know that the crash occurs somewhere in the `Initialize` routine, you only need to trace traps that are called by `Initialize`. A convenient way to isolate the traps to trace is by setting a breakpoint with a break action.

A break action is a SADE command or a set of SADE commands that are executed when a target application encounters a breakpoint. In this case, you can set a breakpoint on `Initialize` and set up a break action instructing SADE to trace all traps. Then you can resume program execution. You can do all of this by executing this command line:

```
Break Initialize.(0) Begin; Trace all traps; End; Go;
```

Note these points about the command line:

- You can place more than one command on a single command line, but you must separate the commands with semicolons.
- There is no semicolon after `Initialize.(0)` because the next command, `Begin`, is not a separate command but part of the `Break` command; it is the first command in the break action.
- The `Begin...End` construct is used to group commands; in this case it signifies the beginning and end of a break action (`Trace all traps`). The command line works if you leave out `Begin...End`, but these commands make the break action explicit.
- There is only one execution command—`Go`—in the command line. SADE can process only one execution command—`Go`, `Step`, or `Target`—at a time.

- *Note:* Another way of doing this is to use the address range option with the `Trace` command by entering this command:
  ```
  Trace All Traps from Initialize.(0)..Initialize.(30).
  ```

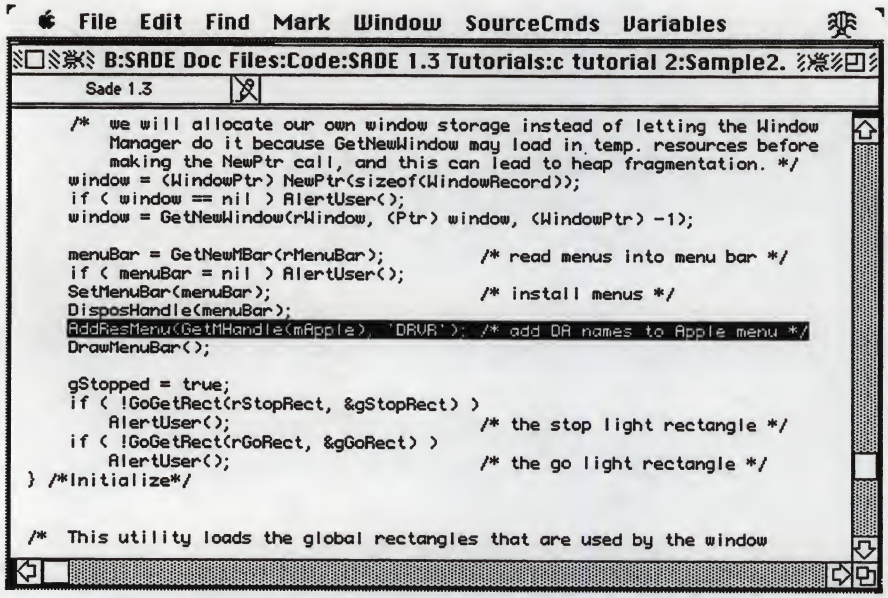
The output on the SADE WorkSheet is:

```
TRACE: Trap _InitGraf called from Initialize.(2)+$0004
TRACE: Trap _InitFonts called from Initialize.(3)
TRACE: Trap _InitWindows called from Initialize.(4)
TRACE: Trap _InitMenus called from Initialize.(5)
TRACE: Trap _TEInit called from Initialize.(6)
TRACE: Trap _InitDialogs called from Initialize.(7)+$0004
TRACE: Trap _InitCursor called from Initialize.(8)
TRACE: Trap _EventAvail called from Initialize.(10)+$000A
TRACE: Trap _EventAvail called from Initialize.(10)+$000A
TRACE: Trap _EventAvail called from Initialize.(10)+$000A
TRACE: Trap _GetTrapAddress called from SYSENVIRONS+$000A
TRACE: Trap _GetTrapAddress called from SYSENVIRONS+$0012
TRACE: Trap _SysEnvirons called from SYSENVIRONS+$0020
TRACE: Trap _GetTrapAddress called from NGETTRAPADDRESS+$000C
TRACE: Trap _GetTrapAddress called from GETTRAPADDRESS+$0004
TRACE: Trap _PurgeSpace called from PURGESPACE
TRACE: Trap _NewPtr called from NEWPTR+$0004
TRACE: Trap _GetNewWindow called from Initialize.(23)+$000C
TRACE: Trap _GetNewMBar called from Initialize.(24)+$0006
TRACE: Trap _SetMenuBar called from Initialize.(27)+$0002
TRACE: Trap _DisposHandle called from DISPOSHANDLE+$0004
TRACE: Trap _GetMHandle called from Initialize.(29)+$0006

Bus Error in "Sample2"
```

The last line labeled `TRACE:` in this output shows that the last statement which Sample2 executed before the bus error occurred was `Initialize.(29)` (as explained in Chapter 3, "Language Overview," the SADE identification number `Initialize.(29)` is a reference to the 29th statement in the `Initialize` function). To see the source code for the statement with this identification label, highlight `Initialize.(29)` and choose Show Selected Routine from the SourceCmds menu. SADE then opens the source file and highlights the statement of interest, as shown in Figure 4-6.

■ **Figure 4-6** Tracing a bus error



In the statements preceding the highlighted statement, the variable `menuBar` appears four times. It thus seems reasonable to say that `menuBar` is a key variable in this group of statements (also note that the last call made before the program crashes is `GetMHandle`, which returns a handle to a menu):

```
menuBar = GetNewMBar(rMenuBar);              /* read menus into menu bar */
if ( menuBar = nil ) AlertUser();
SetMenuBar(menuBar);                         /* install menus */
DisposHandle(menuBar);
AddResMenu(GetMHandle(mApple), 'DRVR');
DrawMenuBar();
```

To see the value of the `menuBar` variable type `menuBar` in the WorkSheet window and press the Enter key. SADE returns this error message:

```
### Could not find "menuBar" as a program symbol
```

As explained in the previous tutorial, the reason for this error message is that `menuBar` is not a local variable or known global variable. In this case `menuBar` belongs to a function higher up in the call chain, so you can try qualifying it with the name of the containing routine in addition to the variable name:

```
Initialize.menuBar
```

SADE then displays this message:

```
### The variable, "menuBar", is in a register and cannot be referenced
### except in its own frame
```

The situation is not completely hopeless yet.To see the value of `menuBar`, you need to be stopped in the function where it is defined. Kill the target, and restart it. Then set a breakpoint on `Initialize.(29)` and resume program execution, by issuing these commands:

```
Kill;
Target 'Sample2' OnEntry Begin;
      Go Til Initialize.(29)
End;
```

- *Note:* The `onEntry` option allows you to execute these commands by highlighting all of them and pressing Enter. The `onEntry` option tells SADE what to do when it is reentered. In the above example the Target command launches the program, SADE is reentered, then SADE sets a temporary breakpoint at `Initialize.(29)` and restarts the target, which after a short while hits the break at `Initialize.(29)`.

  If you execute the commands together without using `onEntry`, SADE attempts to set the breakpoint before launching the program, which it cannot do.

  If you execute the commands, `Kill`, `Target`, `Break` and `Go` one at a time, you do not have to use the `onEntry` keyword.

Now when you try to obtain the value of `menuBar`, SADE displays:

```
menuBar = ^^Byte(NIL)
```

If you are familiar with Macintosh programming, you know that trying to use a NIL handle in a program is virtually certain to result in an address error. The trace of the `Initialize` routine that you carried out earlier in this tutorial showed that that the last Toolbox call made before Sample2 crashed was `GetMHandle`. Another well-known fact about Macintosh programming is that if the `GetMHandle` call cannot find a menu resource it fails. In this case, the handle `menuBar` has a NIL value; it is not pointing to a valid menu resource.

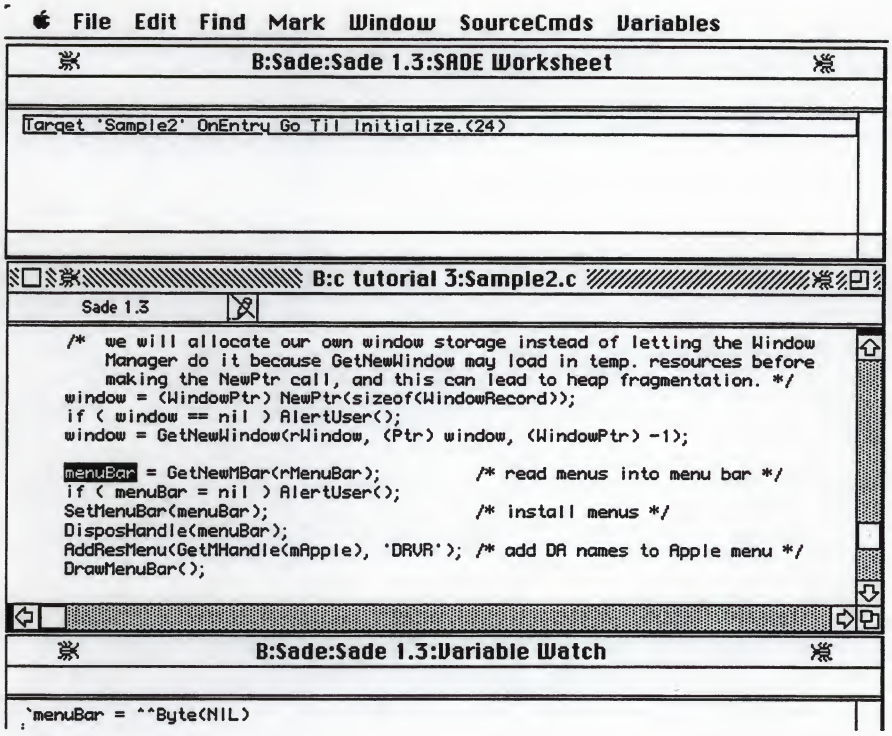
Now you need to determine which statement assigned a NIL value to `menuBar`. You'll need to set a breakpoint sooner than `Initialize.(29)`, because the value of `menuBar` is already NIL at that point. To determine where to set a breakpoint, you can look at the trace output that SADE generated; it lists the Toolbox routines that the program calls. The output shows that `Initialize.(24)` calls `GetNewMBar` ; if you set a breakpoint at `Initialize.(24)`, select `menuBar` as a watch variable, and step through the program, you'll see what value `GetNewMBar` assigns to `menuBar`.

Again, Kill the target, target it again, and go till `Initialize.(24)`, by executing the command:

```
Target 'Sample2' OnEntry Go Til Initialize.(24)
```

Now select `menuBar` as a variable to watch ; highlight `menuBar` in the Worksheet window or another window, and choose Add Watch Variable from the Variables menu. SADE opens the Variable Watch window, which displays the value of `menuBar`. However, the Variable Watch window may be behind other open SADE windows. To make the Variable Watch window visible, choose Tile Windows from the Window menu. SADE then displays all open windows. The next statement to be executed is highlighted in the source window, and the value of `menuBar` is displayed in the Variable Watch window. Assuming you only have three windows open, your display probably now looks something like the one in Figure 4-7.

■ **Figure 4-7** Tiled windows



Step the program once. The value displayed in the Variable Watch window shows:

```
`menuBar = {PStr}^^
```

Step the program again, and the value of menuBar in the Variable Watch window changes back to NIL:

```
`menuBar = ^^Byte(NIL)
```

The last statement executed was:

```
if ( menuBar = nil ) AlertUser();
```

Chances are that this line was meant to check for a NIL value and alert the user. The error is that the assignment operator, =, should be the comparison operator, ==. Thus, the statement should be:

```
if ( menuBar == nil ) AlertUser();
```

To finish the debugging session, issue the Kill command from the SADE Worksheet:

```
Kill
```

To quit SADE, use Quit from the File menu, or issue the Quit command:

```
Quit
```

You can then launch MPW, fix the bug, and execute the corrected program.

Chapter 5 **Pascal Tutorial Examples**

This chapter provides two sample debugging exercises for programs written in Pascal. See the previous chapter for sample C applications.

The tutorial in Chapter 1 shows how to target and debug a simple program using SADE menu commands. The first tutorial in this section introduces additional menu commands and shows how to perform similar operations—such as targeting an application to debug and setting breakpoints—using scriptable commands. The second tutorial contains valuable information on qualifying variables and procedure references.

Sample applications

The applications introduced in this chapter, Sample1 and Stuff, are in the SADE Tutorials folder. The first is a variation of the MPW Sample application into which an intentional bug has been introduced. Sample itself has limited functionality; it switches a traffic light between red and green. As a sample application, it demonstrates how to initialize some of the most commonly used Toolbox managers and how to write a program that operates successfully under MultiFinder and handles desk accessories.

The bug that has been introduced into Sample in Pascal Tutorial 1 is easy to spot: when you click the mouse, the traffic light doesn't switch from red to green.

The MoreStuff application in Pascal Tutorial 2 has no functionality at all. Essentially, it is a collection of variable and function definitions that demonstrate how you must qualify variables and procedure references in SADE.

Keep these points in mind about debugging these sample applications:

- The applications have been successfully compiled and linked. SADE cannot debug an application that won't compile and link.
- The applications were compiled and linked with the `-sym on` option to the Pascal and Link commands. This option generates a symbol file that SADE requires to debug an application. Whenever you want to use SADE to debug a program, you must compile and link with `-sym on`.

Pascal Tutorial 1

This tutorial demonstrates how to perform basic debugging operations using the SADE command language. It shows you how to:

- set a directory and target the application to debug
- set breakpoints and tracepoints
- step through the application

By working through this tutorial, you'll learn how to track down the source of a bug.

Setting the directory and targeting a program

Before you can target Sample1, you must set the default directory to the PExamples folder that is inside the SADE 1.3 Tutorials folder. If SADE is not set to the correct directory, SADE cannot automatically open the necessary source files during a debugging session.

If you select the Target command from the File menu, SADE sets the current directory to that of the selected program. If you debug the same program repeatedly, you may save time by writing and reusing two SADE commands, `Directory` and `Target`, which perform the same operations. Type the following lines in your worksheet and press Enter:

```
Directory 'hdName:SADE:SADE Tutorials:PExamples:P tutorial 1:'
Target 'Sample1' using 'Sample1.sym'
```

Replace *hdName* with the name of the volume that contains the tutorials. Also, if you have placed the tutorials in another directory, be sure to specify the correct pathname to the Sample1 tutorial in the directory command.

For precision's sake, this example explicitly identifies the symbol file name as well as the name of the program to debug. As a rule, you can leave out the symbol file name because SADE automatically looks for the default name *programName*`.sym`. You are required to specify the symbol file name (with the `using` option) only if you have given the symbol file a different name, if the program and its symbol file are in different directories, or in special cases, such as debugging an MPW tool, where the target application is the MPW Shell and the symbol file is that of the tool.

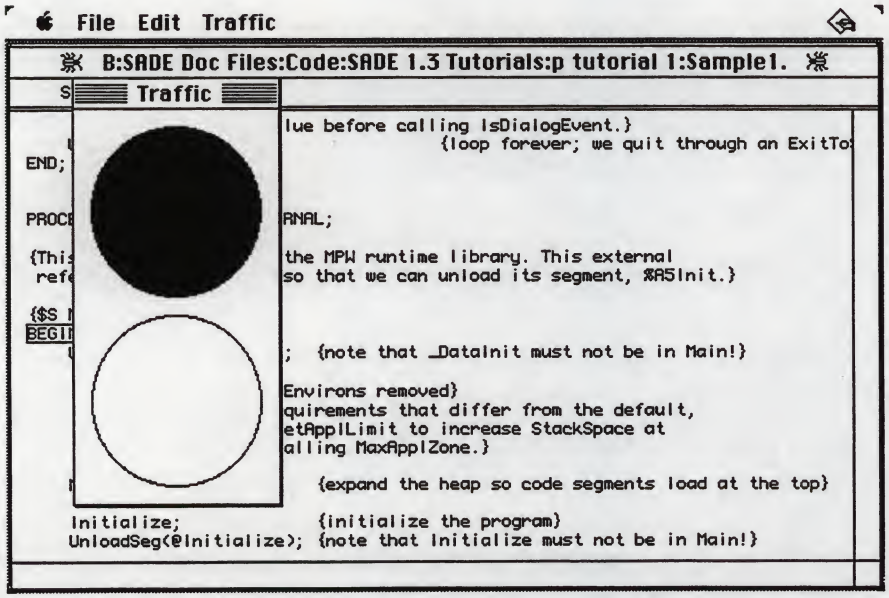
◆ *Note:* In this tutorial, the Sample1 application has only one source file. However, many applications have multiple source files. If the source files reside in different directories, you can use the `SourcePath` command to list all the directories that SADE must search to find the source files. (For more information on `SourcePath`, see Part II, "Command Reference.")

The `Target` command identifies a program to debug, launches the program, and stops at the first statement of the main program. SADE also highlights that statement. To continue the execution of the program, you can do one of the following:

- select Go from the SourceCmds menu
- use the Go command's keyboard equivalent, Command-P
- issue the `Go` command from the SADE Worksheet window

The Sample1 application and its menu bar appear as shown in Figure 5-1.

■ **Figure 5-1** Sample1 application



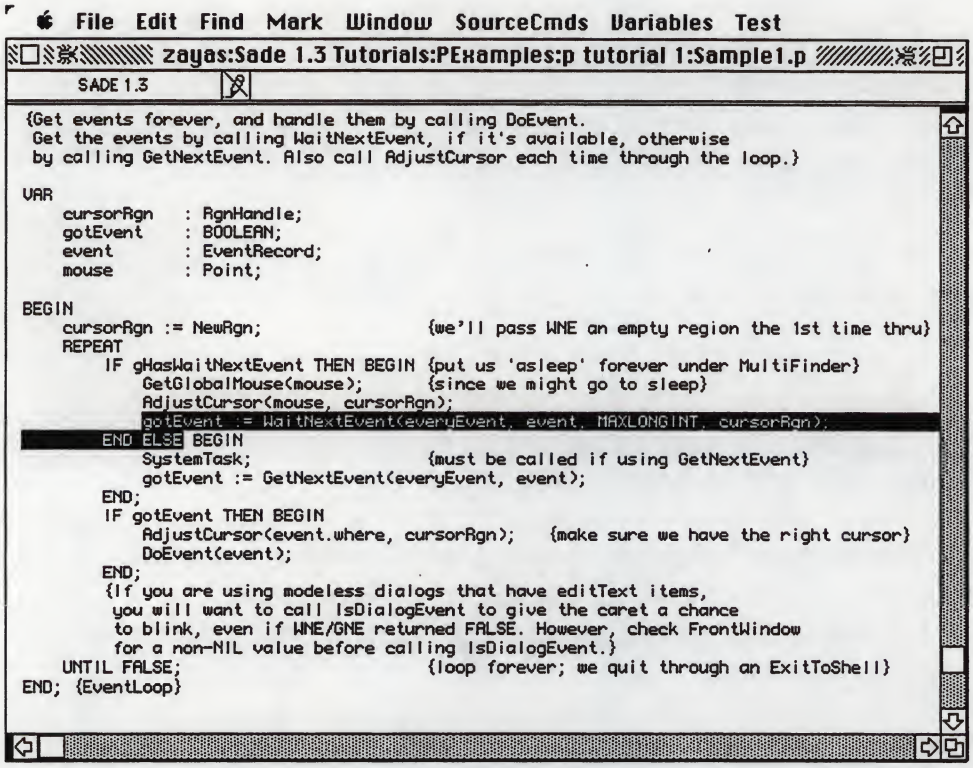
Sample1 has one simple function: to switch the traffic light between red and green (if you have a black-and-white monitor, the top circle is darkened when red, and the bottom circle is darkened when green). In versions of the program that do not contain a bug, you can switch the traffic light between red and green by clicking either circle or by choosing the Green Light or Red Light command from the Traffic menu. If you experiment with Sample1, you will discover that you can switch the lights by choosing the Green Light or Red Light menu command, but clicking inside the circles does not change the light.

- *Note:* If you quit the application while experimenting, you must target Sample1 again before proceeding with this tutorial because quitting removes Sample1 as the target.

Looking at the source file and making a guess

Now that you know there is a bug in the Sample1.p program, how do you go about finding it? The first step is to take a look at the source code. To do that, you can type the SADEKey combination (Command-Option–keypad period) to suspend the execution of Sample1 and bring its source file window to the front. The SADEKey combination suspends an application the next time the application calls a `WaitNextEvent`, `GetNextEvent`, or `EventAvail` routine. When SADE opens the source file, it highlights the statement that called the event trap—in this case, `WaitNextEvent`, as shown in Figure 5-2.

■ **Figure 5-2** Targeting an application



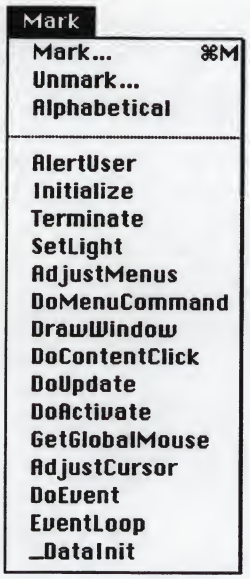
To begin tracking down a bug, you must guess its cause based on what you know about the program. Two significant facts about this application provide a clue to the source of the bug:

- Clicking either circle does not work.
- Selecting the Red Light and Green Light menu commands works as expected.

A solid guess is that something is wrong in a routine that handles mouse clicks.

Therefore, it might be a good idea to search for routines that have the word click in their names. You could use the menu command Find (under the Find menu) to search for the word click. However, SADE recognizes markers placed in a file by the MPW Shell. For your convenience, all of the routines in the source file Sample1 have been marked, and you can see a list of the current markers in the file by pulling down the Mark menu, as shown in Figure 5-3.

■ **Figure 5-3** The Mark menu



One routine on this menu, `DoContentClick`, has a name that suggests it might handle mouse clicks in the Traffic window. So choose `DoContentClick` from the list in the Mark menu. SADE then brings the source file window to the front and highlights `BEGIN` in the `DoContentClick` routine. Figure 5-4 shows the code for the DoContentClick routine.

■ **Figure 5-4** The DoContentClick routine

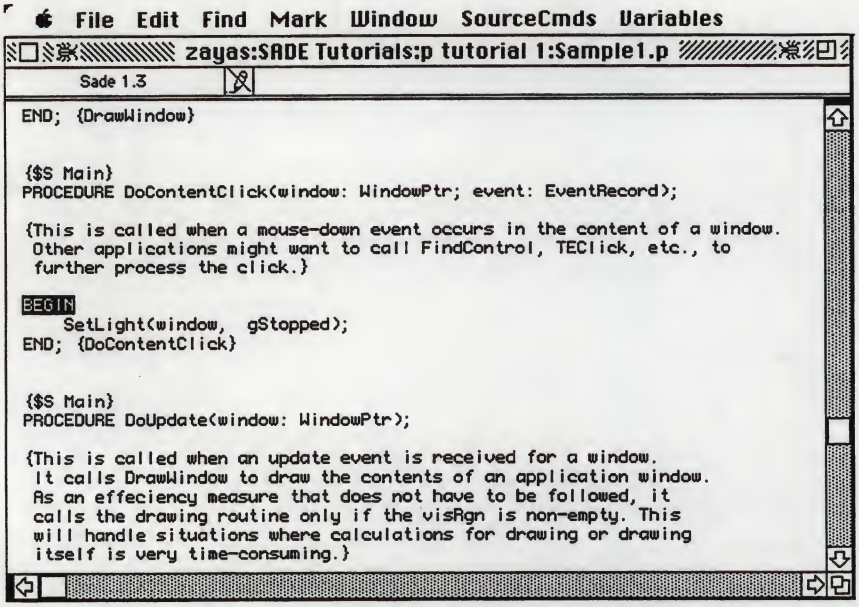


Figure 5-4 appears to confirm that the assumption about `DoContentClick` handling mouse clicks is correct. The comment at the beginning of the `DoContentClick` routine states that the routine handles mouse-down events.

Setting a breakpoint

The strategy for finding the bug in Sample1 is first to verify that `DoContentClick` handles mouse clicks inside the circles. If it does, you can suspend the program at `DoContentClick` and then step through it one source statement at a time to see why clicking the mouse inside the circles doesn't switch the traffic light between red and green.

To test `DoContentClick`, set a breakpoint in `DoContentClick` and run the program to see if the application breaks (suspends execution) when you click the mouse inside the circles.

You can set either a temporary or permanent breakpoint. A temporary breakpoint is one that SADE removes after it has been used once; a permanent breakpoint stays in effect until you remove it or kill the target program. When you set a breakpoint using a menu command, SADE places the breakpoint on the statement corresponding to the location of the insertion point. However, make sure that the insertion point is positioned on or after the `BEGIN` keyword in a routine; otherwise, SADE does not have adequate information to set a breakpoint.

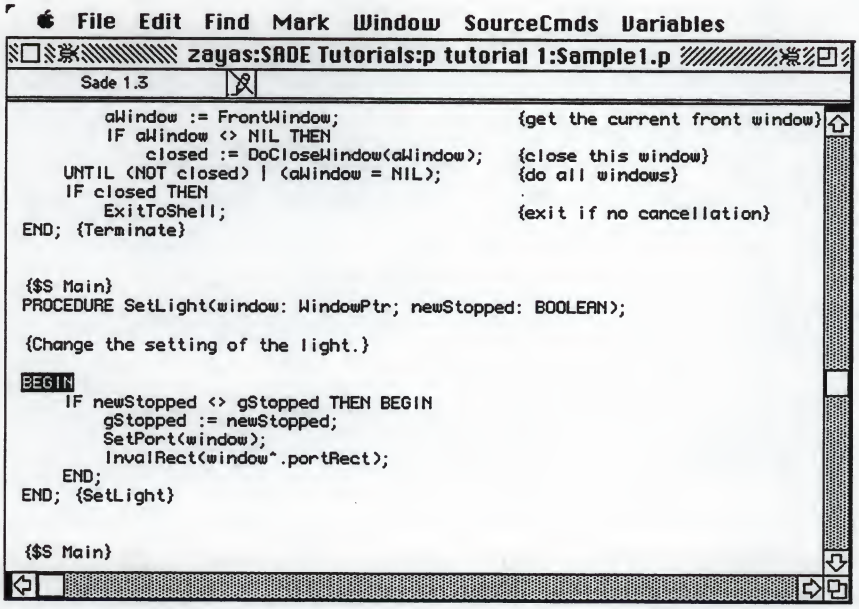
You can set a temporary breakpoint by choosing Go Til from the SourceCmds menu or typing the keyboard equivalent, Command-Option-P. Go Til sets a temporary breakpoint and resumes execution of the program. Then, if you click anywhere within the box surrounding the circles in the picture of the traffic light, SADE suspends execution of Sample1. SADE brings the source file's window to the front and highlights the first statement in the `DoContentClick` routine.

With the execution of Sample1 suspended, you can step through the program by choosing Step from the SourceCmds menu or typing the keyboard equivalent, Command-L. After the first step is complete, SADE highlights the following statement:

```
SetLight(window, gStopped);
```

`SetLight` is definitely a procedure you want to look at. However, if you issue the Step command now, SADE steps over this procedure. So this time choose the Step Into menu command (or type Command-Option-L) to step into the `SetLight` routine. SADE then places the insertion point at `BEGIN` in the SetLight procedure, as shown in Figure 5-5.

■ **Figure 5-5** The Step Into command



The gStopped and newStopped variables

Step twice more, and you'll see that SADE executes the first statement in this procedure but skips the rest of the statement. To understand why this happened, note the `IF` clause that follows the BEGIN keyword in Figure 5-5:

```
IF newStopped <> gStopped THEN BEGIN
```

Apparently, SADE skipped the block of code that follows this `IF` cause because the condition set by the `IF` clause was not met. In other words, a variable named `newStopped` was equal to a variable named `gStopped`, and that prevented the next block of code from being executed.

You can learn more about the variables `gStopped` and `newStopped` by doing a little investigation.

The gStopped variable

As a comment in the heading section of the Sample1 program explains, `gStopped` is a global variable that determines whether the traffic light is red or green; it is red if `gStopped` is `TRUE`, and green if `gStopped` is `FALSE`.

The newStopped variable

The `newStopped` variable is local to the `SetLight` procedure, so you need to examine the `SetLight` routine to see how it is used. You could display the SetLight routine by choosing its name from the Mark menu. But SADE also offers some other useful navigation tools. These include the Show Selected Routine command in the SourceCmds menu and the built-in SADE procedure `AddrToSource`. The Show Selected Routine command and the `AddrToSource` procedure are described in Part II, "Command Reference."

To use the `AddrToSource` procedure, type and highlight `SetLight` in the Worksheet and choose the Show Selected Routine menu command, or execute this command:

```
AddrToSource(SetLight, 1)
```

SADE then places the insertion point on the `BEGIN` statement of the `SetLight` procedure and brings the source window to the front.

As explained in Chapter 3, "Language Overview," SADE uses an identification code called a SADE identification number to refer to statements in a routine. A SADE identification code consists of the name of a routine and a number that is assigned to each statement in the routine. The routine itself is given the identification number *routineName.*`(0)`, the first statement in the routine has the identification number *routineName.*`(1)`, and so on. For example, if you type and highlight `EventLoop.(11)` in the Worksheet and then choose Show Selected Routine, SADE highlights the following statement in Sample1.p:

```
IF gotEvent THEN
```

When this statement is displayed, you can highlight it and choose the Statement Select Is? menu command in the SourceCmds menu to verify that the highlighted statement is indeed `EventLoop.(11)`.

- *Note:* If you are navigating through the source file when SADE has suspended the application at a particular point, keep in mind that you are not changing the location of the program counter when you move the insertion point around.

  SADE uses the program counter to keep track of where the program is suspended. To see the statement currently pointed to by the program counter, you can choose the In What Statement? menu command from the SourceCmds menu. SADE then highlights the statement at which the application is suspended.

  If you type `PC` in the Worksheet and press Enter, SADE returns a decimal representation of the address where the program is suspended. You can obtain the symbolic representation of the program counter—that is the SADE identification number of the source statement at which the program is suspended—by executing the function `Where` as follows:

  ```
  Where(PC)
  ```

  SADE responds:

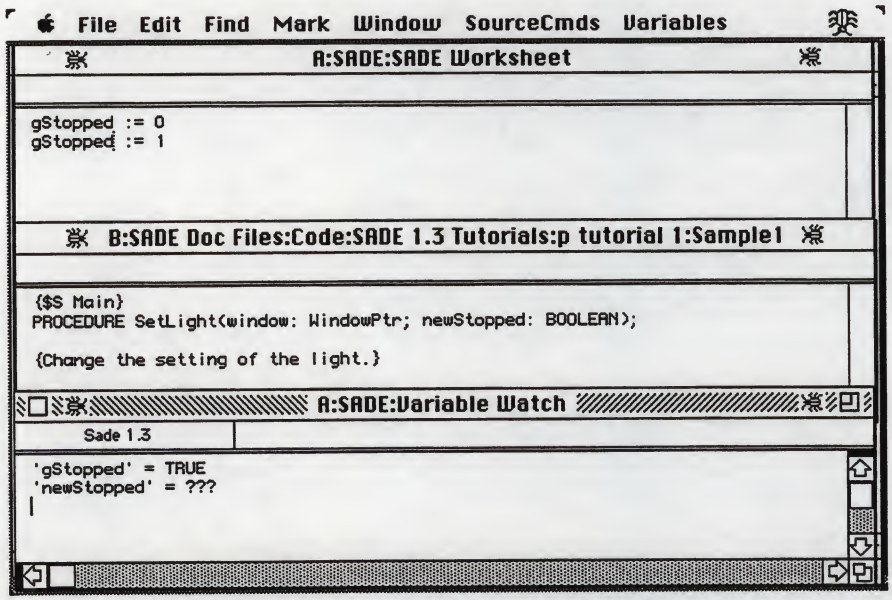
  ```
  SetLight.(6)
  ```

Watching a variable

You can watch the values of `gStopped` and `newStopped` change values as you step through the Sample1 program by using the Add Watch Variable menu command. Highlight either variable's name in the Sample1 source file and then choose the Add Watch Variable menu command from the Variables menu. SADE then opens the Variable Watch window and displays the value of the variable you have selected. Then do the same thing for the other variable. SADE also displays the value of that variable in the Variable Watch window. (If SADE displays a dialog warning you that the `newStopped` variable has not been defined yet, disregard the warning and set `newStopped` as a watch variable anyway.)

When SADE opens the Variable Watch window, there are three overlapping windows on the screen. You can clean up your display by choosing Tile Windows from the Window menu. SADE then places the three windows in a neat horizontal pattern on the screen, as shown in Figure 5-6. This display allows you to type text in the Worksheet and watch the values of your two variables change, while you also watch the source statements that change the values of the variables.

■ **Figure 5-6** Tiled windows



To watch the values of `gStopped` and `newStopped` change values as you step through the Sample1 program, you can put a breakpoint in the DoContentClick routine. To set a breakpoint using a menu command, place the insertion point on the first statement in DoContentClick (on the call to SetLight) and choose the Break command from the SourceCmds menu. To set a breakpoint at the same point in the program from the SADE, Worksheet, execute this command:

```
Break DoContentClick.(1)
```

When you have set a breakpoint in `DoContentClick`, resume execution of the application by choosing the Go command under the SourceCmds menu. Next, noting that the traffic light is red, click inside one of the circles. SADE then suspends execution of the program and highlights `DoContentClick.(1)`. In the Variable Watch window, SADE now displays these values for `gStopped` and `newStopped`:

```
newStopped = Undefined/Out of scope
gStopped = TRUE
```

SADE displays `Undefined/Out of scope` instead of the value of `newStopped` because `newStopped` is out of scope; execution of the program is suspended outside the `SetLight` procedure in which `newStopped` is defined. The value of `gStopped` is `TRUE` because the light is red.

Now choose the Step Into command from the SourceCmds menu (or type its keyboard equivalent, Command-Option-L), to step into the `SetLight` procedure. The value of `gStopped` remains `TRUE` and `newStopped` now has a value of `FALSE`. However, notice that SADE displays the following warning message:

```
Note: Parameters and local variables aren't valid yet at this statement.
```

- *Note:* In the code generated by the MPW compilers, parameters and local variables are normally addressed as an offset of the register A6. In addition, the first instruction in most procedures is a LINK instruction which sets up A6. Until this instruction is executed, SADE would reference wrong memory locations if it tried to display local variables.

Since the variables are invalid at this statement, step once again. At this point, the next statement to be executed is as shown:

```
IF newStopped <> gStopped THEN
```

As shown in the Variables window, `newStopped` and `gStopped` have these values:

```
newStopped = TRUE
gStopped = TRUE
```

Clearly, if the `newStopped` and `gStopped` variables have the same value, a conditional statement that depends on their values being different will not be executed. Step the program one more time and you will see that this is the case; SADE skips over the code that changes the color of the light and moves on to the last statement in the `SetLight` routine.

Execute the program again and change the color of the traffic light to green by selecting the GreenLight command from the Traffic menu. Then click in the area of the circles again. Once again, SADE breaks in the `DoContentClick` routine. Step Into the `SetLight` routine again and notice the values of the variables again match —in this case, they are both `FALSE`—so the color of the light does not change.

It now seems very likely that if `newStopped` and `gStopped` have different values when the `SetLight` routine is executed, the traffic light changes color. To test this hypothesis, run the program again. If the light is not green, choose the Green Light menu command to change it to green. Then click in the uncolored circle, and when SADE suspends execution of the program, step into the `SetLight` routine. Then step through the program until this statement is highlighted:

```
IF newStopped <> gStopped THEN
```

Now the `gStopped` and `newStopped` routines both have a value of `FALSE`. Therefore, click in the SADE Worksheet and set the value of `newStopped` to `TRUE` as follows:

```
newStopped := 1
```

Step four more times and you'll see that SADE executes each of the following statements in turn instead of skipping them as it did previously:

```
gStopped := newStopped;
SetPort(window);
InvalRect(window^.portRect);
```

Note that the value of `gStopped` has changed to `TRUE`. Run the program and you'll see that the light has changed to red. If you wish, you can test the hypothesis again, but starting with a red light instead of a green light and assigning a value of 0 instead of a value of 1 to `newStopped`. However you already have enough information to know where the solution lies. As the next-to-last line of this code fragment shows, the `DoContentClick` routine is passing the value of `gStopped` as the value of `newStopped` when it calls the `SetLight` routine:

```
{$S Main}
PROCEDURE DoContentClick(window: WindowPtr; event: EventRecord);

{This is called when a mouse-down event occurs in the content of a window.
 Other applications might want to call FindControl, TEClick, etc., to
 further process the click.}
```

```
BEGIN
      SetLight(window,  gStopped);
END; {DoContentClick}
```

Therefore, `SetLight` always sets the `newStopped` variable to the value that `gStopped` has. Thus, `gStopped` and `newStopped` always have the same value.

For `SetLight` to work properly, `newStopped` and `gStopped` must have different values, as you have demonstrated. Therefore, `DoContentClick` should pass `NOT gStopped`, instead of just `gStopped`, to the `SetLight` routine. Thus, the last three lines of `DoContentClick` should be changed to read:

```
BEGIN
      SetLight(window,  NOT gStopped);
END; {DoContentClick}
```

Now that you know what the bug in Sample1 is, you can carry out the following tasks:

1. Kill the Sample1 application by choosing Kill from the File menu.
2. Quit SADE by typing Command-Q.
3 Launch MPW, fix the bug in Sample1.p, and rebuild the Sample1 application.

Pascal Tutorial 2

This tutorial shows how to work with program symbols that are outside the current scope of the target program. Before you start the tutorial, you should be familiar with the material on qualifying variables that is presented in Chapter 3, "Language Overview."

The application to be debugged in this tutorial is Stuff, which is compiled from two files: Stuff.p and MoreStuff.p. These files are in the P Tutorial 2 folder within the SADE Tutorials folder. Along with the sources are two more files: Stuff.make, which provides the compile and link commands to build the application, and PTutorialScript, a SADE script that provides all of the necessary commands for running this tutorial.

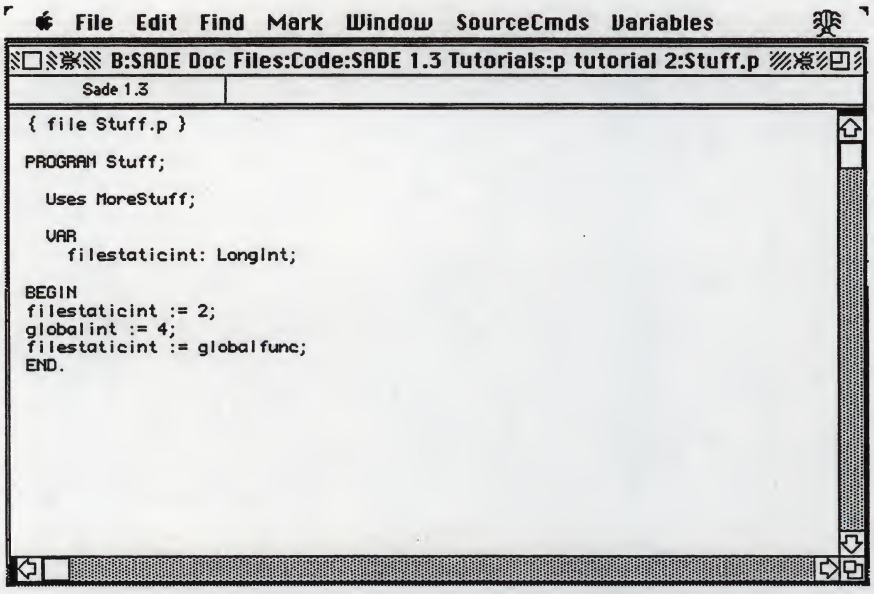
About this tutorial

Stuff is a simple program which illustrates how to access variables and procedures which are outside the current program scope.

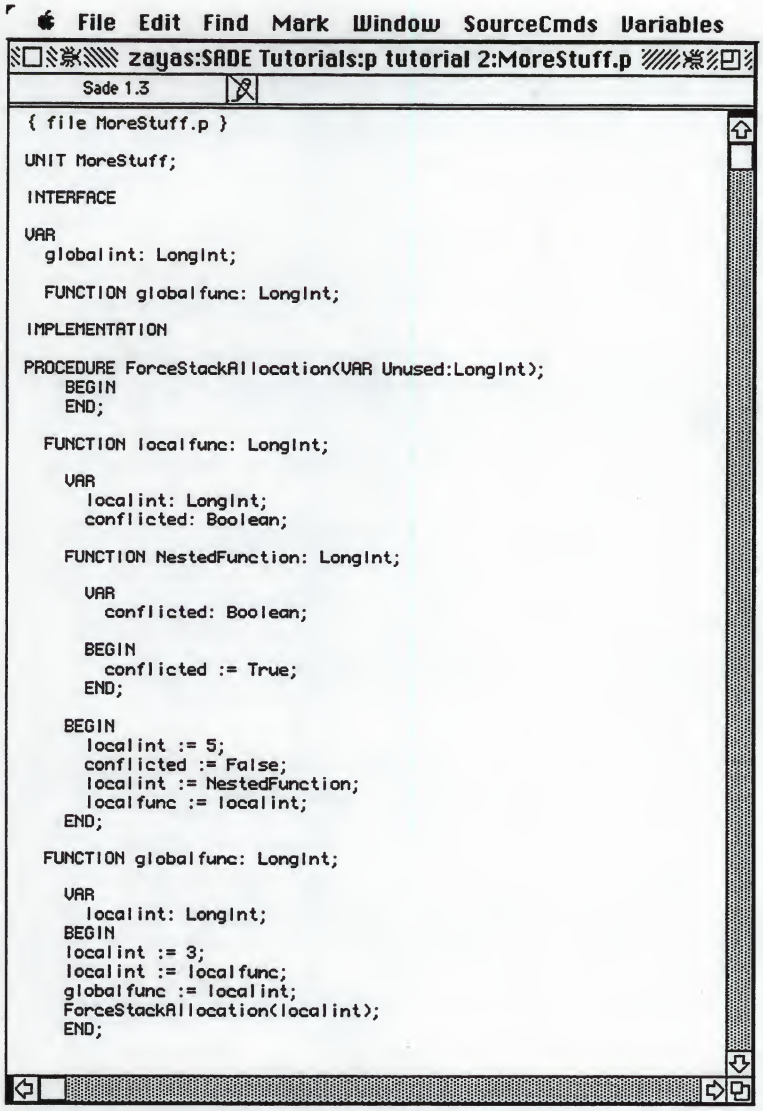
The Stuff program includes functions named `globalfunc`, `localfunc`, and `NestedFunction`; and variables named `localint`, `globalint`, `filestaticint`, and `conflicted`.

The Stuff.p program is shown in Figure 5-7. The MoreStuff.p program is shown in Figures 5-8 and 5-9.

- **Figure 5-7** The Stuff.p listing



■ **Figure 5-8** The MoreStuff.p listing



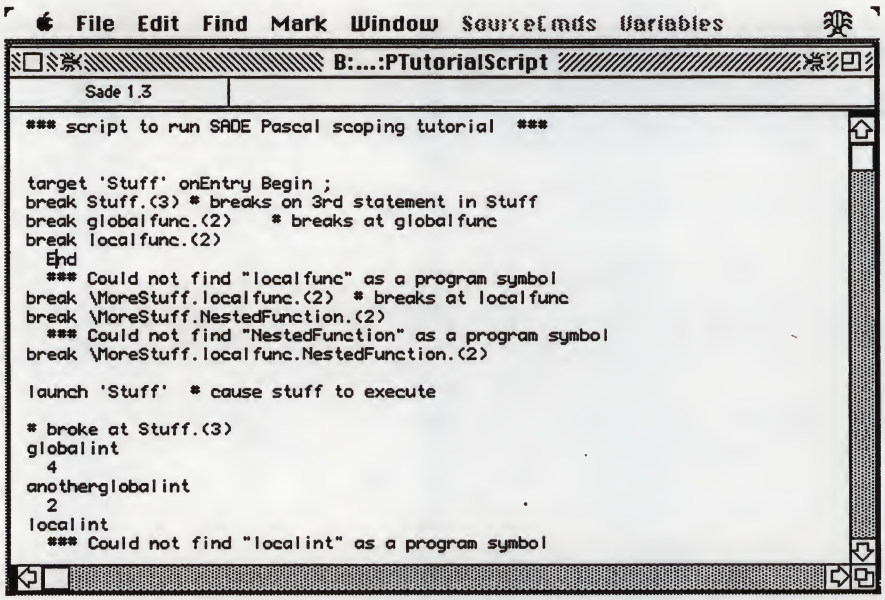
This tutorial describes a hypothetical debugging session using Stuff as a target program. The tutorial illustrates:

- when and how to qualify variables that are local to a file or a function so that they can be accessed from other files or functions
- how SADE handles global variables

Using the PTutorialScript file

You can make the exercise easier by using the file PTutorialScript, which contains all the commands executed in the tutorial. To use this file, simply open it, find the SADE command or series of commands that you want to execute and copy the text you have selected to the Worksheet. Then execute your command or series of commands from the Worksheet. The first part of the PTutorialScript file is shown in Figure 5-9.

■ **Figure 5-9** The PTutorialScript file



Working with symbol scope

To begin this tutorial, execute the following group of commands. The commands attempt to set breakpoints on Stuff, `globalfunc`, and `localfunc`.

```
Target 'Stuff' onEntry Begin;
      Break Stuff.(3)          # breaks on 3rd statement in Stuff
      Break globalfunc.(2)     # breaks at globalfunc
      Break localfunc.(2)
End
```

When you execute this series of commands, SADE sets breakpoints as requested on the third statement in Stuff, and on `globalfunc`. However, SADE cannot find `localfunc` as a program symbol because it is local to MoreStuff.p. Also, SADE cannot find NestedFunction because it is local to `localfunc`. Therefore, SADE displays this message:

```
### Could not find "localfunc" as a program symbol
```

Since SADE cannot find `localfunc`, you must qualify it as follows:

```
Break \MoreStuff.localfunc.(2)
```

◆ *Note:* The scoping of variables and the use of the backslash character are explained in the section "About symbols" in Chapter 3, "Language Overview."

When you have qualified `localfunc` and `NestedFunction`, resume execution of the application by choosing Go from the SourceCmds menu.

When you resume program execution, SADE suspends program execution in the third statement of Stuff.p. At this point, SADE can display the values of `globalint`, which is a global variable, and `filestaticint`, which is local to Stuff.p. To obtain the values of these two variables, just enter their names in the SADE Worksheet window as follows:

```
globalint
  4
filestaticint
  2
```

However, SADE cannot find `localint`, which is local to two different functions. It appears both in the `globalfunc` routine, where it is named `globalfunc.localint`, and in the `MoreStuff.localfunc`.routine, where it is named `\MoreStuff.localfunc.localint`. SADE recognizes both versions of the `localint` variable, but cannot provide a value. Instead, when you try to obtain the value of `localint` by entering its name in the Worksheet window, SADE says it cannot find the variable as a program symbol. If you try to qualify `localint` by informing SADE that it can be found in `globalfunc`, `localfunc`, or even in `\MoreStuff.localfunc`, those attempts also fail:

```
localint
### Could not find "localint" as a program symbol
\MoreStuff.globalfunc.localint
### The variable, "localint", is A6 based and its procedure is not
### in the call chain
\MoreStuff.localfunc.localint
  ### The variable, "localint", is A6 based and its procedure is not
  ### in the call chain
```

The reason SADE displays the message stating that `localint` "is A6 based and its procedure is not in the call chain" is that the compiler has stored the variables on the stack, offset from the register A6. SADE can only return the value of variables within the current call chain. The best way to see the current call chain is to examine the stack using the Show Stack command from the Variables menu.

Continue executing the program and the program should break at `globalfunc.(2)`. Now `localint` (in `globalfunc`) is within the current name scope and the variable can be displayed.

```
localint
3
```

Execute the program once more and the program should break at `localfunc.(2)`. Now, the value of `localint` inside of `localfunc` can be displayed. In addition, since `globalfunc.localint` is within the current call chain, its value too may be displayed.

```
localint
5
globalfunc.localint
3
```

- *Note:* The `ForceStackAllocation` routine does nothing; its only purpose is to provide `globalfunc` with a destination for the parameter `localint`. Because `localint` is passed as a variable parameter instead of a value parameter, the compiler stores `localint` on the stack instead of in a register, which makes `localint` accessible from routines called by `globalfunc`.

You can terminate this debugging session by choosing Kill from the File menu or by executing the `Kill` command from the SADE Worksheet. Then you can exit SADE by choosing Quit from the File menu.

Chapter 6 **Special Debugging Cases**

This chapter describes the special considerations that apply when you are debugging one of the following:

- Object Pascal and MacApp code
- C++ code
- an MPW tool

For more details on these and other special subjects, see the text provided in the New User Worksheet that comes with SADE.

Debugging Object Pascal and MacApp code

SADE supports MacApp methods. Within methods, you can make references to the instance variables of the object without qualifying those references by `SELF`. You can also make these free references to instance variables in C++ member functions.

If you are debugging Object Pascal or MacApp code, be certain that the SADE variable `BreakIfNoSource` is set to 0, the default setting. Setting `BreakIfNoSource` to 0 causes the Step Into command to treat Object Pascal and MacApp method dispatches as indivisible operations: Stepping into a method call breaks at the first instruction of the method. Keep in mind that when SADE steps into something without statement information, it executes your code 2 to 3 orders of magnitude slower than normal, so it may appear that SADE has hung when in fact it is just temporarily running more slowly. Setting `BreakIfNoSource` to 1 makes SADE break when no source information is available for the program counter.

To step into a method, use the StepInto menu item from the SourceCmds menu (with Source vs. Asm checked in the SourceCmds menu) or the Step Line Into command and options.

When you want to debug MacApp code, list the directory path to the MacApp source files with the `SourcePath` command. To avoid stepping through MacApp's method dispatch code, assemble MacApp's `≈.a` files with `-sym off`. Otherwise, the `BreakIfNoSource` command does not do much good.

To debug Object Pascal code that does make use of MacApp, always link your code with `-opt on`. If you don't, SADE cannot operate.

SADE knows only the leaf names of program source files. Therefore, if there are source files in different directories with the same leaf name, the `AddrToSource` function opens the first file it finds and processes the directory paths specified by the `SourcePath` command, which may or may not be the correct source path. If SADE finds the wrong file, change the order in which you pass source paths to the `SourcePath` command—or remove the path that contains the errant file.

Debugging C++ code

By compiling C++ programs with `-sym on` or `-sym full`, you can minimize the mangling that C++ does to field names and local variables, allowing you freer access to instance variables. This means that inside the body of a member function, you can see the member variables without having to qualify their names fully using the construction *this->*. Just select the member variable name and chose Show Value from the Variables menu.

You can also see the value of the implied argument *this*. Just type *this* on the worksheet and press Enter. SADE responds with a line in this format:

```
^TNothingWindow($01603CA4)
```

The current symbol file format does not include information about class static variables, so when you display a class object, the static member variables are not displayed. To display these variables, you need to qualify their names fully, in this fashion:

```
TNothingApp:fgTheApp
```

You can set breakpoints on all member functions of a class. For example, these two commands set breakpoints and tracepoints, respectively, on all member functions of TClass:

```
Break Class TClass
Trace Class TClass
```

These two commands set breakpoints and tracepoints on all overloaded functions with the name *member*:

```
break overload member
trace overload member
```

Debugging an MPW tool

Debugging an MPW tool is not much different than debugging any other program, but there are several important considerations to keep in mind.

When you link an MPW tool, make sure to specify the file type and creator; for example:

```
Link  -t 'MPST' -c 'MPS ' -sym on
```

After you launch SADE, set the directory to the location of the tool's source files; for example:

```
Directory "HD:MPW:Examples:CExamples:"
```

Target the MPW Shell, target the Shell (not the tool name), and specify the symbol file for the tool. For example, this command targets the Count application in the SADE CExamples folder:

```
Target "HD:MPW:MPW Shell" using "Count.SYM"
```

When you have executed this command, you can debug a tool in the same way that you would debug any application. When you are finished debugging a tool, you can remove the tool as a debugging target *but leave MPW running* by executing the `KillTool` command:

```
KillTool
```

The `KillTool` command shuts down the tool you are debugging, but leaves MPW running, so you can continue to work in MPW if you wish.

You can also kill a tool but leave MPW running by executing the Kill menu command from the File menu. If a tool is being debugged, the Kill menu command calls `KillTool` .

If you want to terminate the debugging of a tool and also quit MPW so that you can debug a stand-alone application, execute the Target command with an empty parameter field, as follows:

```
Target " "
```

Alternatively, you can execute the Untarget command:

```
Untarget
```

Appendix A **Summary of Commands and Built-in Functions**

This appendix summarizes the SADE commands and built-in functions. See Part II, "Command Reference," for a complete description of all the SADE commands and built-in functions.

Commands

This section lists the SADE commands. Commands are listed by function.

File commands

`Close`	Closes a file
`MoveWindow`	Moves windows
`Open`	Opens a file
`Redirect`	Redirects standard output
`Save`	Saves a file
`SizeWindow`	Resizes a window
`WindowSize`	Sizes new and zoom windows

Application control commands

`Directory`	Displays or changes the current directory
`Kill`	Kills an application
`SADEKey`	Defines a key for entering SADE
`SourcePath`	Sets search path for source files
`Target`	Selects target program for debugging

Menu and Alert commands

`Addmenu`	Creates a menu or adds menu items
`Alert`	Displays an alert box
`Beep`	Generates tones
`Deletemenu`	Deletes menus or menu items

Heap commands

`Heap`	Displays heap information
`Heap check`	Checks heap consistency
`Heap totals`	Displays heap summary

Resource commands

`Resource`	Displays the resource map
`Resource check`	Checks the resource map

SADE execution commands

`Abort`	Stops break action and cancels pending commands
`Execute`	Executes a script
`Quit`	Quits SADE
`Shutdown`	Shuts down (with restart option)
`Stop`	Stops break action execution and executes pending commands

Breakpoint and tracepoint commands

`Break`	Sets breakpoints
`List`	Lists processes, breakpoints, tracepoints
`Trace`	Sets tracepoints
`Unbreak`	Removes breakpoints
`Untrace`	Removes tracepoints

Program flow control commands

`Go`	Starts execution
`Step`	Steps through code

SADE programming commands

`Begin...End`	Groups commands
`Cycle`	Continues execution at conditional test of current looping construct
`For...End`	Loops with a control variable
`Func...End`	Defines a SADE function
`If...End`	Conditionally executes commands
`Leave`	Leaves current looping construct
`Loop...End`	Repeats commands until `Leave`
`OnEntry`	Sets commands for SADE entry
`Proc...End`	Defines a SADE procedure
`Repeat...Until`	Conditionally loops with end test
`Return`	Returns from a SADE procedure or function
`While...End`	Conditionally loops with beginning test

SADE variable commands

`Define`	Declares a SADE variable
`Undefine`	Removes a SADE variable, macro, function, or procedure

Special-purpose display commands

`Disasm`	Disassembles code
`Dump`	Displays unstructured memory in hexadecimal
`Stack`	Displays stack frames

Miscellaneous commands

`Case`	Changes case sensitivity
`Find`	Searches for a target
`Help`	Displays help information
`Macro`	Creates a macro
`Printf`	Sends formatted output to file or window
`Version`	Displays SADE version number

Built-in functions

This section lists the SADE built-in functions. The functions are listed alphabetically.

`AddrToSource`	Displays the source statement corresponding to an address
`Concat`	Concatenates string expressions
`Confirm`	Displays confirmation dialog box
`Copy`	Copies all or part of a string
`Eval`	Evaluates a string as an expression
`Find`	Searches for a pattern in memory
`Length`	Returns the length of a string
`NaN`	Converts to a SANE 10-byte extended value
`Request`	Displays a request dialog box
`Selection`	Returns the text of the current selection in a specified window
`SizeOf`	Returns the size of a variable or type
`SourceToAddr`	Returns the address corresponding to a selected source statement
`Timer`	Provides timing-related functions
`TypeOf`	Returns the type of an expression
`Undef`	Determines whether a SADE parameter or variable has been initialized
`Where`	Returns the symbolic representation of an address

Appendix B **Customizing SADE**

This appendix explains how SADE is configured for standard operation and how you can customize SADE operation to suit your own needs.

The easiest way to customize SADE is to examine how things are done in the file SADEStartup. Copy the script you want to change and modify it. If you want to make a permanent change, it is advisable that you put it in the file SADEUserStartup and not in SADEStartup. This guarantees that you can always restore original SADE behavior and you do not have do this customization every time you get a new version of SADE.

The SADEStartup file

Each time you launch SADE, it executes the SADEStartup file. This file contains SADE commands, functions, procedures, and variable definitions that configure SADE for source-level debugging. In particular, the SADEStartup file creates the SourceCmds and Variables menus and modifies the File menu. If you study the way each of the features is implemented, you can change them to suit your own needs.

For instance, when you interrupt your program, SADE brings the source window up as the front most window, and does not display an alert box. These actions are controlled by definitions in SADEStartup:

```
define SourceInFront := 1      #source brought up as frontmost window
define BreakAlert    := 0      #don't display an alert at break
```

You can change the values of these variables from the SADE Worksheet if you like by entering:

```
SourceInFront := 0             #source brought up behind command window
BreakAlert    := 1             #display an alert at break
```

The new values remain in effect for that debugging session only. The next time SADE is launched, the definitions in SADEStartup are executed again, restoring the default display.

The SADEUserStartup file

If you want to redefine a certain feature indefinitely, or to add new features, you can put new definitions into the SADEUserStartup file. Initially empty, SADEUserStartup is executed by SADEStartup. You can change the definitions in SADEStartup directly, but it's safer to change SADEUserStartup so you don't accidentally edit something useful in SADEStartup.

In SADEUserStartup you can enter SADE commands and define your own procedures and functions. For instance, you can create new menus for executing commands by using `AddMenu` commands.

The OnEntry command

The `OnEntry` command specifies what to do each time SADE is entered. `OnEntry` can take a single command, a group of commands, a procedure, or a function as an argument. (There is also an `onEntry` option for each of the SADE execution commands, as explained in Chapter 3, "Language Overview.")

In SADEStartup, the procedure `StandardEntry` is passed to the `OnEntry` command. `StandardEntry` performs a number of useful actions. It identifies where and why your program code is interrupted and the kind of interrupt that has occurred. It also provides for source display of the current program counter (PC) on entry into SADE. If the source cannot be displayed, SADE displays the cause of the interruption, the instruction at the program counter at the time the interruption occurred, and the name of the suspended program. This procedure lists the numeric codes that correspond to the different types of interrupts and system errors that could occur and specifies how to handle them. For example, SADE prints an error message for exceptions such as bus errors and address errors.

You can use the `OnEntry` command to specify additional or different actions, but you need to know what you're doing. Each time the `OnEntry` command executes, the actions specified by the previous `OnEntry` command are replaced. You'll probably want to use the `StandardEntry` procedure as a model, modify it to suit your needs, and pass it to `OnEntry` in the SADEUserStartup file.

One thing you can do is add to the list of exceptions in `StandardEntry`. For example, you can put an interrupt directly into your source code by making a call to the Toolbox SysErr routine and passing an error number (be sure to pick an unused error number in the range 129 to 32,5111). In `StandardEntry` (or a similar procedure called by `OnEntry`) you can list the exception (error number) as part of a conditional statement and specify how you want SADE to handle it. When your program encounters the exception, it passes control to SADE, which acts according to your instructions.

Appendix C **Formal Grammar**

This appendix formally describes the SADE language.

Formal language description

expression
    term
    term op term
If term is a *string*, *op* must be a relational operator

term
    symbol
    constant
    function

symbol
    SADE symbol
    program symbol
    system symbol

SADE symbol
    command (one of)

```
Abort  AddMenu  Alert  Beep  Begin...End  Break  Case  Close
Cycle  DeleteMenu  Directory  Disasm  Dump  Execute  Find
For...End  Func...End  Go  Heap  Help  If...End  Kill  Leave
List  Macro  MoveWindow  Open  Printf  Proc...End  Quit
Redirect  Repeat...Until  Resource  Resource check  Return
SADEKey  Save  Shutdown  SourcePath  Stack  Step  Stop
Target  Trace  Unbreak  Untrace  Version  While...End
```

    predefined variable (one of)

```
ActiveWindow  Arg[n]  Date  DisAsmFormat  Exception  Inf
NArgs  ProcessId  TargetWindow  Worksheet
```

    user-defined variable
        symbol name

program symbol
    \unit[*.procedure...*]*.variable*
    \unit.procedure[*.procedure...*] [*.(index)*]
    procedure[*.procedure...*]
    procedure[*index*]*.variable*
    [`]*variable*

Note that for *procedure*`[` *index*`]` `.` *variable* the brackets are for an array type syntax (they are not the syntax brackets meaning optional) and you must enter them. The index number indicates how many times the variable has been called by a recursive procedure, so you can see the value of the variable at a particular iteration of the procedure.

system symbol
[Δ]*predefined variable*
    register
    toolbox call

variable
    firstcharacter one of:
        `A - Z  a - z  _  %`
    subsequent character: up to 62 of
        `A - Z  a - z  _  %  $  #  @`

To use characters in a variable name that are not in the regular character set described above, precede them with the Backslash (`\`) escape character.

numeric constant
    decimal: (up to 32 bits)
        `0 - 9`
    `$`*hexadecimal* (up to 32 bits)
        `0 - 9, A - F, a - f`
    `%`*binary* (32 bits left padded)
        `0 - 1`
    floating-point

string constant up to 254 of
    '*ASCII character*'
    "*ASCII character*"

Within double quoted strings there is special processing with the Backslash `\` and Delta ∂ escape characters. You can precede two hex characters or three decimal characters with the Backslash or Delta character (`\$`*xx* or `\`*nnn*), which SADE interprets as their ASCII equivalent. This enables you to put control codes in string constants. You can also add three control codes like so, `\n` (line feed); `\f` (form feed); `\t` (tab).

A CString is up to 254 characters terminated by a null character. A PString is a length byte followed by up to 254 characters.

function
    built-in

```
AddrToSource  Concat  Confirm  Eval  Find  Length  NaN
Request  Selection  SizeOf  SourceToAddr  Timer  TypeOf
Undef  Where
```

user-defined
variable[(*arg*, ...)]

variable
value
address
type coercion
type(*expression*)
type is program defined or one of these SADE base types

```
Boolean  Byte  CChar  Char  Comp[utational]  CString  Double
Extended  Extended12  Float  Int  Integer  Long  LongInt
PascalChar  PChar  PString  Real  Short  SignedLong
SignedLongInt  Single  Str255  String  Unsigned
UnsignedByte  UnsignedChar  UnsignedInt  UnsignedLong
UnsignedLongInt  UnsignedShort  UnsignedWord  Word
```

cast or group operator
(*expression*)

qualify operator
.*expression*

qualify by pointer operator
->*expression*

increment operator
++*variable*
variable++

decrement operator
--*variable*
variable--

trap operator
†*expression*

address of operators
&*expression*
@*expression*

unary operators
+*expression*
-*expression*

arithmetic operators

expression op expression

op is one of

```
+  -  *  /  DIV  ÷  //  MOD  <<  >>
```

shift operators

expression << *expression*

expression >> *expression*

logical operators

expression op expression

op is one of

```
=  ==  <>  ≠  !=  >  <  <=  ≤  >=  ≥  &&  ||
```

Evaluates to true (1) or false (0).

bitwise operators

expression op expression

op is one of

```
&  AND  |  OR  XOR  EOR
```

unary logical operators

op expression

op is one of

```
NOT  ¬  !  ~
```

assignment operator

variable := *expression*

If the variable on the left of an assignment statement is a SADE variable, any type assignment can be made (SADE variables are dynamically typed—they take on the value of the type assigned to them). For an integer, SADE saves the value on the right into the variable on the left. SADE converts a floating point number to extended and then assigns it to the variable on the left.

assignment with operation

variable op= expression

op is one of

```
+  -  *  /  DIV  ÷  //  MOD  <<  >>
```

If the variable on the left is a SADE variable, any type assignment can be made (SADE variables are dynamically typed—they take on the value of the type assigned to them). For an integer, SADE performs the operation on the left and right values and saves the result into the variable on the left. After performing the operation, SADE converts a floating point number to extended and then assigns it to the variable on the left.

arbitrary assignment operator
variable <- *expression*

The size of the value on the right determines how many bytes to move.

pointer operators
variable^
**variable*

range operator
expression . . *expression*

If one expression is a trap the other one must be a trap as well. Floating point numbers are not allowed in either expression.

Part II Command Reference

Appendix D **SADE Commands**

This part of the manual describes SADE commands and built-in functions. The commands and functions are listed together in alphabetical order. The "Type" heading for each entry indicates whether the entry is a command or a built-in function.

There are two major differences between commands and functions. First, all functions return a value, whereas some commands do and some do not. Second, although you cannot use a SADE command as part of an expression, you can use a function in this way. For example, you can take the output returned by a function such as `SourceToAddr` and assign it directly to a variable:

```
myVar := SourceToAddr(TargetWindow)
```

Note that you must enclose the parameter list for all SADE built-in functions in parentheses.

Abort—stops execution of scripts

Syntax `Abort`

Type Command

Description The `Abort` command terminates execution of the current script and returns you to SADE, canceling all pending commands. This means that if current execution is within a structured statement (`Begin...End`, for instance), or if multiple commands are selected, these pending commands are not executed.

To terminate a break action without canceling pending commands, use the `Stop` command.

Example

The way in which `keyDown` and `mouseDown` events are handled by the `EventFilter` procedure in this example illustrates the difference between `Abort` and `Stop`. A `keyDown` event (in the context of this example) does not entail any additional action, so you can use `Abort` to terminate the break action and return to SADE immediately. In the case of a `mouseDown` event, however, the location of the mouse when the break occurs is important; therefore, you use `Stop` to terminate the break action so that the `Printf` command showing the location of the mouse is executed before SADE is reentered.

```
#-- Establish target, suspending application
#-- and reentering SADE at main.(1)

Directory 'VolName:some:path:toProg:'
Target 'MyProg'
Break \MyProg.main.(1)

#-- Break action examines the current event type
#-- (assuming the event record is named myEvent).
#-- For mouseDown, shows coordinates of mouseDown and stops
#-- (quitting break action but executing rest of pending SADE commands)
#-- For keyDown, aborts (quitting break action AND rest of commands).
#-- For any other event, shows event type and continues executing
#-- (by default, break actions end with an implicit Go command)

Proc EventFilter
  Define global mouseAt;
  If myEvent.what = 1 then    # mouseDown event
    mouseAt := myEvent.Where;
    Stop;
  elseif myEvent.what = 3 # keyDown event
```

```
    "keyDown"; Abort;
  else
    Printf "Event type %d: \n", myEvent.what;
    Printf;
  End
End

Begin
  Break _getNextEvent from applzone..applzone^ EventFilter
  Go
  Printf "Mouse down at H: %d\n          V: %d\n", mouseAt.h, mouseAt.v;
End

#-- output

Event type 0:
Event type 8:
Event type 6:
Mouse down at H: 208
              V: 144

#-- With same break action in place, restart target.

Go

#-- output

Event type 6:
keyDown
```

See also Break, Stop

AddMenu—creates a menu or adds menu items

Syntax `AddMenu` [*menuName* [*itemName* [*command*]]]

Type Command

Description The `AddMenu` command lets you create menus and add menu items to execute SADE commands. The parameters *menuName*, *itemName*, and *command* are all string expressions; if they are string constants, you must enclose them in quotation marks.

menuName — Specifies a menu name. If a menu of this name does not exist in the menu bar, a new menu is created.

itemName — Specifies the name of an item in the specified menu. If a menu item of *itemName* already exists, it is replaced; otherwise, a new menu item is created. You can include a keyboard equivalent for the item by listing the key after a slash (`/`) at the end of the string. See the "Examples" section. (Warning: Be careful not to inadvertently specify a keyboard equivalent that is already defined for a different menu item, because the new definition will replace the existing one without any warning.)

command — Specifies a string that is executed when a user clicks on *itemName*. It can be a SADE command (see the first example in the "Examples" section), or something more complex ,such as the name of a procedure that you have defined (see the second example in the "Examples" section).

If you do not specify *itemName* or *command*, `AddMenu` returns the current value from the specified level down. For instance, if you specify *itemName* without any commands, `AddMenu` displays the command that's currently defined for that menu item. If you omit *menuName*, *itemName*, and *commandName* as well, SADE returns the current values for all user-defined menus and menu items.

Examples

This example creates a menu called Debug with two items, Disasm and Code Resources. Disasm disassembles instructions starting at the end of the last disassembly (the default behavior for the `Disasm` command without parameters). Code Resources displays information about code resources. Note that you can invoke Disasm from the keyboard with Command-4.

```
AddMenu 'Debug' 'Disasm/4' 'Disasm'
AddMenu 'Debug' 'Code Resources' 'Heap restype "CODE"'
```

This example defines a procedure called `ShowWhere` and then creates a menu item, Statement Selected Is?, which executes this procedure. `ShowWhere` shows what routine a selected statement belongs to; if there isn't sufficient symbol information, `ShowWhere` displays an error message explaining why it cannot show the routine name.

```
Proc ShowWhere
      Define loc := SourceToAddr(ActiveWindow, 1)
      If typeOf(loc) = 'PString' then
            Alert Concat('Cannot determine address from source: ', loc)
      else
            Alert Concat('At ', Where(loc))
      End
End

AddMenu 'SourceCmds' 'Statement Selected Is?' 'ShowWhere'
```

See also `DeleteMenu`

AddrToSource—displays source statement corresponding to address

Syntax `AddrToSource (`*address*`[, `*Boolean*`])`

Type Built-in function

Description `AddrToSource` displays and selects the source statement corresponding to the specified address. If the source file isn't already open, it is displayed as a read-only window. If the source file is already open as a read/write window, `AddrToSource` changes the window to read-only.

If you omit the optional *Boolean* value or specify 0 (FALSE), SADE displays the source window behind the front most window (most likely the window from which you executed `AddrToSource`). If you specify a nonzero value for *Boolean* (TRUE), SADE brings up the window as the front most window.

`AddrToSource` returns a Boolean value indicating whether it was able to display the source statement (TRUE) or not (FALSE).

See also `SourceToAddr`

Alert—displays an alert box

Syntax `Alert [Beep] message`

Type Command

Description The `Alert` command displays an alert box containing the specified message. The alert box is displayed until you click the OK button.

message	Specifies the message to be displayed. It is a string expression; if it is a string constant, it must be enclosed in quotation marks.
Beep	Generates a sound when the alert box appears.

Example

```
If length(str) > 64 then
  Alert "string longer than expected"
End
```

See also `Beep`

Beep—generates tones

Syntax `Beep [ note [, duration [, level ]] ... ]`

Type Command

Description The `Beep` command produces each given note for the specified duration and sound level. You must separate each note specification (that is each note and its optional duration and level) from the next note by blank spaces or tabs. If you do not specify *note*, SADE produces a standard Macintosh warning beep of predetermined pitch and duration.

note Specifies the note to be produced. Your parameter can be one of the following:

- A number indicating the count field for the square wave generator, as described in the Sound Driver chapter of *Inside Macintosh, Volume II*
- A string in the format [*n*] *letter* [`#` | `b`], where *n* is an optional number indicating the octaves below or above middle C, followed by a letter indicating the note (A–G) and an optional sharp (`#`) or flat (`b`) character

duration Specifies the duration in sixtieths of a second. The default duration is 15 (one-quarter second).

level Specifies the magnitude of the sound. Your parameter can be a number from 0 to 255. The default level is 128.

Example

```
Beep '2C,20,75 2C#,40,150 -1D,60,75'

#-- Play the three notes specified: C, C sharp, and D, for one-third,
#-- two-thirds, and one full second, respectively. The middle note is
#-- twice as loud as the other notes. C and C sharp are two octaves
#-- above middle C. D is one octave below.
```

See also `Alert`

Begin...End—groups commands

Syntax

```
Begin
   commands
End
```

Type Command

Description The `Begin...End` construct allows you to group together or bracket a sequence of commands. You can use this construct to specify a break action consisting of multiple commands or procedure calls.

commands Specifies one or more SADE commands.

Example

This example specifies a conditional break action that depends on the value of the pointer `theStr^`.

```
Break DisplayString.(4) Begin
  str := theStr^   # save value of parameter in variable str
  If str = '***' then
    Stop
  End
End
```

This example shows how to use the `Begin...End` construct to nest `Step` commands.

```
Step onEntry Begin
   Printf "gStopped is %d\n",gStopped
   Printf "newStopped is %d\n",newStopped
   Step onEntry Begin
      Printf "gStopped is %d\n",gStopped
      Printf "newStopped is %d\n",newStopped
   End
End
```

Break—sets breakpoints

Syntax

`Break` *addr*,... [*breakAction*]
or
`Break` *trap* [`from` *addrRange*],... [*breakAction*]
or
`Break` *trapRange* [`from` *addrRange*],... [*breakAction*]
or
`Break all traps` [`from` *addrRange*] [*breakAction*]

Type

Command

Description

The `Break` command sets one or more breakpoints in a target program's code. There are two types of breakpoints: address breakpoints and trap breakpoints. You can follow either kind of breakpoint by a break action, that is, a command (or series of commands) that is executed when the breakpoint is reached.

addr	Specifies a RAM address or symbolic reference. If you use a symbolic reference, the code need not be in memory when you set the breakpoint.
trap	Specifies a trap name or number. You must prefix a trap number with the trap character (†, or Option-T) and a trap name with an underscore character (`_InitGraf`, for example).
trapRange	Specifies a range of traps. You must use the range operator (`..`) between trap names or numbers.
breakAction	Specifies a command or series of commands that is executed when the breakpoint is reached.
`from` *addrRange*	Indicates that a break is to occur only when the trap is called from the specified range.
`all traps`	Sets a breakpoint on every trap.

You can set trap breakpoints on a single trap, a range of traps, or on all traps. In each case you can also specify a memory range so that SADE breaks only when the trap is called from the specified range.

If you want to specify the same trap in multiple `Break` commands, use the `List` command to see what breakpoints have been set. When implementing trap breakpoints, SADE first looks for a Toolbox or Operating System call that matches a specified trap name, checking the address range if one was given. If no match is found, SADE then looks for trap ranges containing the trap. SADE takes the most recently defined range containing the trap, again checking the address range if one was specified.

- *Note:* If you set an address breakpoint on a trap call for which a trap breakpoint has already been set, the address breakpoint is recognized and the trap breakpoint is not.

If you have specified a break action, SADE resumes program execution after executing your break-action commands. There are two ways to suspend program execution as part of a break action. The `Abort` command returns control to SADE immediately, canceling any pending commands. The `Stop` command executes any pending commands and then returns control to SADE.

▲ **Warning** SADE saves the commands you specify in the break action and does not interpret them until reaching the breakpoint. Symbol references that may appear to be correct when you define the break action may be out of scope when the action is actually executed. So make sure that program symbol references are correctly interpreted at the time the breakpoint is reached.

Do not delete the breakpoint as part of a break action. SADE stores the break action with the breakpoint; if you remove the breakpoint, you throw away the break action as well—even though SADE is in the middle of executing it—so the action does not execute properly. ▲

- *Note:* If no break action is specified *and* the execution command (`Go`, for instance) just prior to hitting a breakpoint is part of a structured statement (such as `While...End`), or if multiple commands were selected, the remaining commands are executed after hitting the breakpoint and re-entering SADE.

You can specify multiple breakpoints, separated by commas, with a single `Break` command. If this command includes a break action at the end, the action is applied to all breaks in the list.

Examples

This example shows breaks set on different traps, trap ranges, and traps called from particular memory ranges.

```
Break _GetResource
Break †$A997..†$A9A0
Break all traps from myproc.(1)..myproc.(5)
Break all traps from applZone..applZone^
```

The following example sets multiple breakpoints.

```
Break procB.(1), _LineTo from procA.(1)..procA.(1000), _setPort Begin;
"hit one"; end
```

```
List break
  procB.(1)    # $586398  # processID =5  # has break action
  _LineTo  # processID =5  # †$A891  # called from procA.(1) ..
procA.(83)  # has break action
  _SetPort  # processID =5  # †$A873  # has break action
```

A break set on top of a matching breakpoint replaces the older one. Breakpoints match if they are set on the same address or trap and the called from address range, if there is one, matches.

```
Break DoMenuCommand.(0)
Break _LineTo
List Break
Trap _SetPort (†$A873)
Trap _LineTo (†$A891)
DoMenuCommand.(0)
```

See also `Abort`, `List`, `Stop`, `Trace`, `Unbreak`

Case—changes case sensitivity

Syntax `Case on | off`

Type Command

Description This command turns case sensitivity on and off. By default, case sensitivity is on, which means that when looking up symbols, SADE first performs a case-sensitive lookup. If the symbol isn't found, SADE converts all the characters to uppercase and looks again. When case sensitivity is off, SADE converts all symbols to uppercase before any lookup, speeding up the search process.

`on`	Turn case sensitivity on when looking up symbols.
`off`	Turn case sensitivity off when looking up symbols. SADE converts all characters to uppercase before searching.

Because symbols in C are case sensitive, C programmers must have case sensitivity turned on or SADE will not recognize any of the symbols in the program (except those that are all uppercase). Pascal programmers, however, may want to set case sensitivity off to speed up performance. Note however, that even with case set to on, SADE will find Pascal symbols (no matter what combination of uppercase and lowercase characters you enter) because if SADE cannot find a symbol in a case-sensitive search, it performs a second search after converting all characters to uppercase.

Examples These examples illustrate how `Case on` and `Case off` work by showing the output you get when typing symbols in different combinations of lowercase and uppercase letters.

```
#-- C programmers
Case on
CFunction
#-- output
CFunction.(0)

CFUNCTION
#-- output
### Could not find "CFUNCTION" as a program symbol
```

```
cfunction
#-- output
### Could not find "cfunction" as a program symbol

Case off
CFunction
#-- output
### Could not find "CFunction" as a program symbol

#-- Pascal programmers
Case on
PascalProc
#-- output
PASCALPROC.(0)

PASCALPROC
#-- output
PASCALPROC.(0)
Case off
PascalProc
#-- output
PASCALPROC.(0)
```

Close—closes a file

Syntax `Close [all | windowName]`

Type Command

Description The `Close` command closes the specified file or all files.

windowName	The name of the window (file) to close. *windowName* is a string expression and you must enclose it in quotation marks if it's a string constant.
`all`	Close all windows open in SADE.

If you don't specify any parameters, `Close` closes the target window. Note, however, that the SADE Worksheet file cannot be closed. If the contents of a file have not been saved, a dialog box asks whether they should be.

Examples

In the following example, note that `myFile` requires quotes because it is a string; `targetWindow` does not because it is a predefined SADE variable.

```
Close 'myFile'
Close targetWindow   #no quotes needed--uses SADE variable targetWindow
```

Concat—concatenates strings

Syntax `Concat ([` *string*`,...])`

Type Built-in function

Description `Concat` returns the concatenation of the specified string expressions. If you supply nonstring arguments, `Concat` tries to coerce them to strings. If you specify no arguments, `Concat` returns a null string.

Example

In this example, the `pathName` variable holds the name of a directory path and the `fileName` variable holds a filename. With `pathName` and `fileName` as its arguments, `Concat` returns a complete pathname, which you can use with a command such as `Open`.

```
pathName := 'root:examples:'
fileName := 'sample.c'
Open Concat(pathName,fileName)      # this is equivalent to
                                    # open 'root:examples:sample.c'
```

Confirm—displays confirmation dialog box

Syntax `Confirm ( message[, Boolean] )`

Type Built-in function

Description `Confirm` presents a dialog box containing the specified message and either two or three buttons depending on the value you specify for *Boolean*. Confirm returns a number indicating the button that you press in the dialog box.

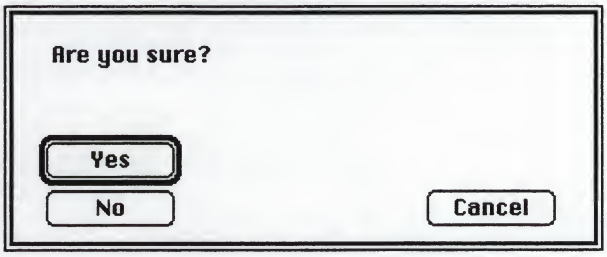
If you omit the optional *Boolean* or specify 0, `Confirm` presents OK and Cancel buttons, and returns 1 or 0 respectively when you press them. If you specify a nonzero value for *Boolean*, `Confirm` presents Yes, No, and Cancel buttons, which return 1, 0, and –1 respectively when you press them.

Example

This example creates the dialog box shown in Figure II-1.

```
confirm('Are you sure?',1)
```

■ **Figure II-1** Confirm dialog box



See also `Request`

Copy—copies string

Syntax `Copy ( string, characterIndex, length )`

Type Built-in function

Description `Copy` returns all or part of the specified string.

Use *characterIndex* to indicate the first character to copy; for example, to start with the third character of the string, specify 3 (it's 1-based).

Use *length* to specify he length of the output string.

Example

In this example, `Copy` strips the hard drive name off the directory path (because the copy begins with the fifth character) and copies the rest of the directory path to the `pathName` variable. You can then open a file by concatenating `pathName` with a filename.

```
directoryPath := 'root:examples:cexamples:'
pathName := copy(directoryPath,5,255)
...
Open Concat(pathName,'sample.c')
```

Note that this example specifies 255 as a number for the length because it is large enough to accommodate long pathnames.

Cycle—continues execution within construct

Syntax `Cycle [if` *Boolean*`]`

Type Command

Description The `Cycle` command causes execution to continue from the conditional test of a `While`, `Repeat`, or `For` construct, or from the beginning of a `Loop` construct.

if *Boolean*	Execute `Cycle` only if the Boolean expression (*Boolean*) evaluates to nonzero; otherwise execution continues immediately following the `Cycle` command.

Example

```
Define testNum := 0
Define endNum := 6
Define cycleMax := 4

Repeat
  Printf "testNum = %d\n", testNum
  testNum := testNum + 1
  Cycle If testNum < cycleMax
  "\nDidn't cycle"
Until testNum = endNum

#-- output

testNum = 0
testNum = 1
testNum = 2
testNum = 3

Didn't cycle
testNum = 4

Didn't cycle
testNum = 5

Didn't cycle
```

See also `Leave`

Define—declares a SADE variable

Syntax `Define [ global ]` *declaration* [,...]

Type Command

Description The `Define` command defines one or more SADE variables. You must define a variable before using it. The variable declaration identifies the name, scope, and (optionally) the initial value of the variable. Separate multiple declarations by commas.

declaration — The name and optional dimension and initial value of the variable. The *declaration* has this form:

name [[*dimension*] [:= *initValue* | = *initValue*]

name — The name of the variable. It must be unique in the current scope unless you declare it global.

dimension — The size of the array if it is a structured variable. It is an expression enclosed in brackets.

initValue — An expression for the initial value of simple types, or for structured types a list of the following form:

([*expr* of] *initValue* , ..)

where the optional of clause allows for replication of a value.

SADE variables are dynamically typed; that is, their type is determined on assignment (and may be changed by new assignments). SADE variables defined as arrays require an index. SADE array variables may contain a heterogeneous set of values; that is, the elements may contain values of different types.

An initial value for simple types may optionally be specified by an *expr* following the assignment operator (:=) or, in this case only, (=). If the declared item is an array, a list of initial values may be specified as the values of the array elements.

The scope of a variable can be either global or local. If a variable is defined outside a procedure (or function), its scope is automatically global. In other words, it is known both inside and outside of any procedures. If a variable is declared inside a procedure, its scope is local unless you specify the `global` option. If a global and a local variable exist with the same name, the local symbol overrides the global symbol.

Redefining global variables replaces the previous definition, with one exception: If the definition is within a procedure, and the new definition matches the existing definition, then the existing definition is retained. For example, when a global variable is defined within a procedure or function and is given an initial value, the initialization occurs only when the variable is actually created. Subsequent invocations of the procedure do not affect the current value of the global variable, and can make use of the value left in the variable by the preceding invocation.

You cannot use the `Define` command within any structured statement. To remove a variable definition, use the `Undefine` command.

Examples

This example defines a five element array, with the first four elements true and the last element false

```
Define global test[5] := (4 of 1,0)
```

This example defines a 30 element array, with the first 29 elements true and the last element false

```
Define arraysize := 30
Define myArray[arraysize] := (arraysize-1 of 1,0)
```

In the next example, note that the definition of `index` in `RotateLeftOneWCarry()` is local to that procedure; otherwise it would be attempting to redefine the global `index` that's used as a `For` loop counter when `RotateLeftOneWCarry()` is called.

```
Proc RotateLeftOneWCarry()
  Define global SADEArray[4] := ("indexed by one", 2, 3.33, "fourth and
last")
  Define index
  Define holder
  For index := 1 to 4 do
    If index = 1 then
      holder := SADEArray[index]
    else
```

```
      SADEArray[index-1] := SADEArray[index]
      If index = 4 then
        SADEArray[index] := holder
      End
    End
  End
End

Define index
Define holder
For index := 1 to 4 do
  RotateLeftOneWCarry()
  For holder := 1 to 4 do
    SADEArray[holder]
  End
  Printf "\n"
End
Undefine RotateLeftOneWCarry

#-- output

2
3.33
fourth and last
indexed by one

3.33
fourth and last
indexed by one
2

fourth and last
indexed by one
2
3.33

indexed by one
2
3.33
fourth and last
```

See also `Undefine`

DeleteMenu—deletes menus or menu items

Syntax `Deletemenu [ menuName[ itemName]]`

Type Command

Description The `DeleteMenu` command deletes menus and menu items.

menuName The name of a user-defined bar menu. If you specify *menuName* but do not specify *itemName*, SADE deletes the entire menu. Because *menuName* is a string expression, you must enclose it in quotation marks if it is a string constant.

itemName An item in the bar menu that you specified with *menuName. ItemName* is a string expression and you must enclose it in quotation marks if it is a string constant.

You cannot delete the standard SADE menus and menu items (the , File, Edit, Find, Mark, and Window menus).

▲ **Warning** If you omit both *menuName* and *itemName*, SADE deletes all user-defined menus, that is, the SourceCmds and Variables menus (which are defined by commands in the `SADEStartup` file) and any menus that you have added in the `SADEUserStartup` file. You can restore these menus by executing SADEStartup and SADEUserStartup. ▲

Example

```
Deletemenu "Special" "Launchapp"
```

See also `AddMenu`

Directory—sets or writes the default directory

Syntax `Directory [` *directoryName*`]`

Type Command

Description The `Directory` command sets the default directory for all SADE file-oriented operations to the specified directory. If you don't specify *directoryName* SADE displays the current default directory.

directoryName A directory name. It follows the same conventions as MPW, that is, you can specify a full or partial pathname. See the MPW manual and the examples below for more information. *directoryName* is a string expression and you must enclose it in quotation marks if it is a string constant.

Examples

In the first example, assume that the current directory is `HD:MPW` and you want to change it to `HD:MPW:Examples`. To do so you can specify a partial pathname,

```
Directory 'Examples'
```

However, assume once again that the current directory is `HD:MPW` but you want to change to a directory, `CExamples`, which is a subdirectory of `Examples`. You can still specify a partial pathname, but you must precede the directory name with a colon,

```
Directory ':Examples:CExamples'
```

If a directory specification contains a colon but does not begin the with a colon, SADE reads it as a complete pathname, as in these two examples,

```
Directory 'HD:MPW'
Directory 'HD:'
```

You can also define macros for your work directories that you use a lot and substitute the macro name for the directory path, as in the next example. Note that the directory string in the macro definition requires two sets of quotation marks.

```
Macro here "'HD:MPW:Example:CExample'"
Directory here
```

See also `Sourcepath`

Disasm—disassembles code

Syntax

`Disasm` [*addr* [*count*]]
or
`Disasm` [*addrRange*]

Type

Command

Description

The `Disasm` command disassembles instructions starting at the location specified by *addr* or *addrRange*. The default behavior when no address is specified is to begin disassembling at the end of the last disassembly. If the value of the program counter has changed since the last disassembly, the program counter (PC) is used as the default starting address. If no range or count is specified, the number of instructions (not lines) disassembled defaults to 20.

addr	The address at which to begin the disassembly.
count	The number of instructions to disassemble. The default, if you do not specify a count, is 20 instructions.
addrRange	The range of addresses for the disassembly. You specify a starting and ending address separated by the range operator (..). See "Examples".

Each line of the disassembly output is divided into three fields: the address of the instruction, the hexadecimal encoding for the instruction, and the assembly code (opcode, operand, and comment). You can modify the presence, order, and format of these fields by changing the value of the built-in variable `DisAsmFormat` (described in Chapter 5). The disassembly output also correlates source statement numbers with the assembly code instructions (see the second example).

Examples

This example disassembles five instructions in standard format, starting at the fourth statement of the `AdjustMenus` routine.

```
Disasm $23B5EE 5

#-- output
```

```
AdjustMenus.(4)
     0023B5EE  2F0B            MOVE.L      A3,-(A7)
     0023B5F0  7004            MOVEQ       #$04,D0
     0023B5F2  3F00            MOVE.W      D0,-(A7)
     0023B5F4  A939            _EnableItem                    ; A939
     0023B5F6  6008            BRA.S       AdjustMenus.(6)    ; 0023B600
```

The next example disassembles the instructions between the first statement of the `AdjustMenus` routine and the fourth statement of that routine.

```
Disasm AdjustMenus.(1)..AdjustMenus.(4)

#--output

AdjustMenus.(1)
     0023B5D2  598F            SUBQ.L      #$4,A7
     0023B5D4  A924            _FrontWindow                   ; A924
     0023B5D6  285F            MOVEA.L     (A7)+,A4
AdjustMenus.(2)
     0023B5D8  598F            SUBQ.L      #$4,A7
     0023B5DA  3F3C 0081       MOVE.W      #$0081,-(A7)
     0023B5DE  A949            _GetMHandle                    ; A949
     0023B5E0  265F            MOVEA.L     (A7)+,A3
AdjustMenus.(3)
     0023B5E2  2F0C            MOVE.L      A4,-(A7)
     0023B5E4  4EBA 02FC       JSR         IsDAWindow.(0)     ; 0023B8E2
     0023B5E8  4A00            TST.B       D0
     0023B5EA  588F            ADDQ.L      #$4,A7
     0023B5EC  670A            BEQ.S       AdjustMenus.(5)    ; 0023B5F8
AdjustMenus.(4)
     0023B5EE  2F0B            MOVE.L      A3,-(A7)
     0023B5F0  7004            MOVEQ       #$04,D0
     0023B5F2  3F00            MOVE.W      D0,-(A7)
     0023B5F4  A939            _EnableItem                    ; A939
```

See also `Dump`

Dump—displays memory

Syntax

`Dump [byte | word | long][` *addr* `[` *count*`]]`
or
`Dump [byte | word | long] [` *addrRange*`]`

Type

Command

Description

The `Dump` command displays memory at the location specified by *addr* or *addrRange*. If no parameter is given, the memory starting at the program counter is displayed. The memory is displayed in hexadecimal and ASCII characters according to the specified format, which may be byte, word, or long. The default format is word.

`byte`	Group the output by bytes.
`word`	Group the output by words (two bytes). This is the default.
`long`	Group the output by long words (four bytes).
addr	The address at which to begin the dump.
count	The number of bytes to dump. The default, if you do not specify a count, is 16 bytes.
addrRange	The range of addresses for the dump. You specify a starting and ending address separated by the range operator (..). See "Examples".

Remember that to dump the value of a variable `myVar`, you must specify `Dump @myVar` (since `Dump myVar` would take `myVar`'s value and use it as an address).

Examples

This example dumps 16 bytes (the default if no count is specified) from the A5 register. The output is grouped in bytes.

```
Dump byte A5

#-- output

$00146A8E 00 14 68 74 FF FF FF FF  00 00 00 00 00 00 00 00  ..ht........
```

This example dumps 40 bytes from the A5 register and groups the output in words.

```
Dump word A5 40

#-- output

$00146A8E 0014 6874 FFFF FFFF  0000 0000 0000 0000  ..ht............
$00146A9E 000E 6C66 0000 FFFF  FFFF FFFF FFFF FFFF  ..lf............
$00146AAE 0001 4EF9 000E 6E00                       ..N...n.
```

This example dumps 40 bytes from the A5 register and groups the output in long words. Note that it uses an address range (from the beginning of A5 to an offset of 40 bytes) to specify the 40 bytes.

```
Dump long A5..A5+40

#-- output

$00146A8E 00146874 FFFFFFFF  00000000 00000000  ..ht............
$00146A9E 000E6C66 0000FFFF  FFFFFFFF FFFFFFFF  ..lf............
$00146AAE 00014EF9 000E6E00  00                 ..N...n..
```

See also `Disasm`

Eval—evaluates string as an expression

Syntax `Eval (`*text*`, [ `*message*`])`

Type Built-in function

Description `Eval` evaluates the text of a string argument as an expression. The function result can be any type, depending on what the expression evaluates to. You can optionally supply a message; if an error occurs, `Eval` returns this message as the function result. If you do not supply a message and an error occurs, `Eval` aborts and reports the error.

Example

This example passes the name of a program variable (`menuId`) to a variable in SADE (`theVar`). SADE uses the `Printf` command to print the name of the variable and `Printf` with `Eval` to print its value. If SADE cannot determine the value of the program variable (for example, because it is out of scope), `Printf` prints `???` (the message part of `Eval`).

```
theVar := menuId
Printf "%t = %t\n", theVar, Eval(Concat('`', theVar), '???')
```

Note that this example uses `Concat` with the backquote character (`` ` ``) to treat `theVar` as a program symbol. The code in this example is similar to code in the `SADEStartup` file that implements the Show Value menu item.

Execute—executes commands in a file

Syntax `Execute` *filename*

Type Command

Description The `Execute` command executes the commands and definitions contained in the specified file. The Execute command can't be used within a structured statement.

filename The name of the file to execute. It is a string expression and must be enclosed in quotation marks if it's a string constant.

Example

In this example, the `Redirect` command creates a file to hold output from SADE. Entering a string echoes the string. That output is redirected to the file, becoming the file's contents. Here the string is a comment and a SADE command to execute the contents of the next file in the chain.

```
Open 'exec1'
Redirect 'exec1'

  '"\n executing exec1 now"'
  "execute 'exec2'"

  Open "exec2"
  Redirect 'exec2'
    "'now executing exec2'"
    "execute 'exec3'"

    Open "exec3"
    Redirect 'exec3'
      '"Done in exec3"'

Redirect pop all
Execute "exec1"

Alert "Try a tile windows here\n∂
   Then look at the worksheet for output"

#-- output

executing exec1 now
now executing exec2
Done in exec3
```

Find—searches for a pattern

Syntax `Find (`*pattern*`,` *address*`,` *length*`[,` *count*`])`

Type Built-in function

Description `Find` looks for a pattern in the memory range specified by a starting *address* and *length.* If you pass zero for *count,* `Find` returns the number of occurrences of the pattern. You can use the *count* parameter to specify which occurrence of the pattern you want; for instance, if you specify 3 for *count,* `Find` returns the address of the third occurrence of the pattern. If you omit *count,* `Find` returns the address of the first occurrence. `Find` returns 0 if it cannot find the target.

Remember that expression values in SADE are long words by default; to specify another size, use a type coercion (see "Examples"). For instance, looks for a long value, and looks for a word value.

Examples

The first example looks for a long value; the second example uses type coercion to search for a word value.

```
Find($ABCD,PC,20)
Find(word($ABCD),PC,20)
```

See also `Find` (command)

Find—searches for a target

Syntax

`Find[count]`*pattern*`[,`*n*`][`*addrRange*`[mask` *mask*`]]`
or
`Find[count]`*pattern*`[,`*n*`] [`*addr*`[` *count*`][mask` *mask*`]]`

or
`Find[same [`*addrRange*`] | [`*addr* `[` *count*`]]]`

Type

Command

Description

The `Find` command searches memory for a specified pattern, which can be either a numeric or string expression. If you specify the `count` keyword, `Find` tells you how many occurrences of the target it found. If you omit the `count` keyword, `Find` displays the address of the first occurrence of the pattern. You can use the *n* parameter to specify which occurrence of the pattern you want; for instance, if you specify 3 for *n*, `Find` returns the address of the third occurrence of the pattern.

`count`	If you use this keyword `Find` tells you how many occurrences of the target string it has found. If you omit it, `Find` displays the address of the first occurrence of the target string.
pattern	A numeric or string expression you wish to find.
n	An integer indicating which occurrence of the target pattern you want; for example, if you specify 3, SADE displays the address of the third occurrence of the pattern. Note that you must precede this number with a comma.
addrRange	An address range in which to limit the search. If you do not specify a range of addresses (or a starting address), the search range is the MultiFinder block containing the application's heap and stack (in other words, all of the memory that belongs to the application).

addr count — An address at which to begin the search. You can also specify how many bytes to search beyond the specified address; otherwise the search goes from the starting address to the end of memory. If you do not specify a starting address or range of addresses, the search range is the MultiFinder block containing the application's heap and stack (in other words, all of the memory that belongs to the application).

`mask` *mask* — A keyword and a numeric or string expression that is logically ANDed with the contents of each memory location before the search comparison is done.

`same` — A keyword that instructs `Find` to search for the next occurrence of the same target pattern as specified in the last `Find` command. You can optionally specify a different address range or starting address and number of bytes for the search. If you used the `count` keyword in the previous `Find` command, you cannot use `same` in the next one; if you do so, SADE returns a message that the pattern cannot be found.

△ **Important** Remember that expression values in SADE are long words by default; to specify another size, use typecasting, as shown in the examples. △

Examples

The `Dump` command in this example displays the contents of the memory range that comprises the examples that follow.

```
Dump $20 $40
$00000020 0027 A002 0027 A00A  0040 1F52 0027 A012  .'...'...@.R.'..
$00000030 0027 A01A 0027 A032  0040 113C 0027 A02A  .'...'.2.@.<.'.*
$00000040 0040 113C 0040 113C  0040 113C 0040 113C  .@.<.@.<.@.<.@.<
$00000050 0040 113C 0040 113C  0040 113C 0040 113C  .@.<.@.<.@.<.@.<
```

This example looks for the number of occurrences of the string, $113C, starting at address $20 and searching the next $40 bytes. The string has been cast to word size; otherwise the default search would be for a long word and nothing in the range matches $0000113C. The search finds nine occurrences of the string.

```
Find COUNT (word)$113C $20 $40
9
```

This example searches the range $20 to $96 for the string $113C and displays the address of the first occurrence of this string. The search string has been typecast to word size using C style casting.

```
Find (word)$113C $20..$96
$0000003A
```

This example searches the range $20 to $40 for the string $113C and displays the address of the second occurrence of this string. The search string has been typecast to word size using Pascal style casting.

```
Find word($113C), 2 $20 $40
$00000042
```

This example searches the range $20 to $96 for the string $0000113C and displays the address of the first occurrence of this string. A mask has been used

```
find $0000113C $20..$96 MASK $0000ffff
$00000038
```

This example finds the same target string as the last example, except the memory range has been changed.

```
Find SAME $40..$50
$00000040
```

See also `Find` (built-in function)

For...End—loops with a control variable

Syntax

```
For clause[do]
  commands
End
```

Type

Command

Description

The `For...End` construct provides looping with a control variable. The enclosed commands are executed until the control variable has taken on each successive value in the range expressed by *clause*.

clause Determines the value of the control variable. There are three forms that you can use for *clause*:

var := *expr* `to` *expr*
The commands are executed and the control value (*var*) is incremented once for each integer value in the range.

var := *expr* `downto` *expr*
The commands are executed and the control value (*var*) is decremented for each integer value in the range.

var := *expr* , ...
The commands are executed until the control variable has taken on the value of each of the listed expressions.

You must declare the control variable *var* before using it, and you cannot specify an array variable.

You can nest `For...End` constructs; you may also use them within other flow-control constructs, as well as in break actions. The control variable may be modified within the body of the loop (but cannot, of course, be shared between nested constructs).

Example

```
Define var := 0
Define outerLooper, syncopation
For outerLooper := 3 downto 1 do
  "\n"
  For syncopation := "one","two","three", var
    Printf "%d -- " , outerLooper
    syncopation
    var := var + 1
  end
End

#-- output

3 -- one
3 -- two
3 -- three
3 -- 0

2 -- one
2 -- two
2 -- three
2 -- 4

1 -- one
1 -- two
1 -- three
1 -- 8
```

Func...End—defines a SADE function

Syntax

```
Func name [( argName,... )]
   commands
end
```

Type Command

Description SADE functions are delimited by the `Func...End` construct. The last statement to be executed must be a `Return` command specifying a return value. The type of a function is not specified in the definition but rather takes on the type of the value returned. (Thus functions are not limited to returning results of a single type.)

name	The name of the function.
argName	An optional list of parameter names separated by commas. You can enclose the parameter list in parentheses if you wish, but if you do so, you must also use parentheses when calling the procedure (and vice versa).
commands	The SADE commands and language elements that compose the function. The last statement to be executed must be a Return command specifying a return value.

SADE functions use conventional calling notation, with the function name followed by a list of parameters enclosed by parentheses. Function parameters are handled in the same fashion as procedure parameters, and the predefined SADE variables `Arg` and `NArgs` may be used. (See the description of the `Proc` command for an example using these variables.) User-defined functions may be called anywhere an expression is allowed.

Example

This function factors a number and displays the total.

```
Func fact(n)
  If n <= 1.0 then
    Return 1.0
  else
    Return n * fact(n-1)
  End
End
```

See also `Proc`, `Return`

Go—resumes execution

Syntax `Go [til` *addr*`,...] [onEntry` *commands*`]`

Type Command

Description The `Go` command resumes program execution at the current program counter. If you specify the keyword `til`, SADE sets a temporary breakpoint at the specified address(es).

`til` *addr* Sets a temporary breakpoint at the specified address or addresses. When the breakpoint is encountered, SADE is reentered and the breakpoint is removed. Note that if you specify multiple breakpoints with the keyword `til`, SADE removes all breakpoints when any of them is reached. If the address is in ROM, SADE informs you that it can't set a breakpoint in ROM.

`onEntry` *commands* Specify actions to occur upon returning to SADE after resuming program execution. See the examples.

Use `onEntry` to specify an action to occur when SADE is reentered following resumption of the program—for example, to display the value of a variable. If you do not use `onEntry` but execute the `Go` command as part of a group of commands (in a script file or highlighted in the Worksheet) SADE evaluates the other commands first before executing `Go` regardless of the order in which you place the commands in the script or in the Worksheet.

△ **Important** You cannot execute a `Step` command following `Go` in a script or as part of a highlighted group of commands unless you use the `onEntry` keyword. With `onEntry` you can tell SADE to execute a `Step` command after returning to SADE following the Go command. (See the last example.) △

Examples

The first example resumes operation of the program and breaks at `\OtherCompilationUnit.myProc.(1)`.

```
Go til \OtherCompilationUnit.myProc.(1)
```

Assume that the program is suspended and a breakpoint has been set within the procedure that initializes the values of the variables `gstopped` and `newstopped`. The following code fragment resumes program execution, stops at the breakpoint, and displays the values of `gstopped` and `newstopped`.

```
Go onEntry Begin
   Printf "gStopped is %d\n",gStopped
   Printf "newStopped is %d\n",newStopped
end
```

See also `Step`, `Stop`

Heap—displays heap information

Syntax `Heap[display][addr][ blocktype ]`

Type Command

Description The `Heap` command displays information about the specified heap. If you do not specify a heap, `Heap` displays the heap pointed to by the global variable `theZone`.

`display`	Display information about the specified heap. This manual has divided the `Heap` command into three parts—display, check, and summarize—for clarity of presentation, though in fact it is one command with three options. `Display` is the default if you don't specify any options, so you can omit it if you wish.
addr	The beginning address of a heap. If you omit this parameter, `Heap` displays the heap pointed to by the global variable `theZone`.
blocktype	You can specify one of the following *blocktypes* to limit the display to a particular type of block:
`purge[able]`	Limits the display to purgeable blocks.
`nonreloc[atable]`	Limits the display to nonrelocatable blocks.
`reloc[atable]`	Limits the display to relocatable blocks.
`free`	Limits the display to free blocks.
`lock[ed]`	Limits the display to locked blocks.
res[ource]	Limits the display to resources.
`restype '`*type*`'`	Limits the display to the specified resource *type*.

- *Note:* The resource type is case-sensitive and you must enclose it in single quotation marks (`'MENU'`, for example).

By default, `Heap` displays this information:

- a dot if the object is locked or nonrelocatable
- the address of the beginning of the block and the block length
- the block type: H=relocatable, P=nonrelocatable, F=free
- the address of the master pointer if it's a relocatable block
- block attributes (Flags): L=locked, R=resource, P=purgeable
- for standard toolbox data structures, a description of the structure
- for a resource, the resource type and ID, the reference number of the file it's in, and the resource name

Example

The command in this example displays heap information for `'MENU'` type resources.

```
Heap restype 'MENU'

#-- output

BlkAddr    BlkLength Typ MasterPtr  Flags RType   RId RFRef  RName
$00316590  $00000098  H  $0031452C    R   MENU   1000 $0584  "File"
$00316838  $00000050  H  $00314528    R   MENU   1001 $0584  "Edit"
$00316888  $000000F4  H  $00314524    R   MENU   1002 $0584  "Log"
```

See also `Heap check`

Heap check—checks heap consistency

Syntax `Heap check` [*addr*]

Type Command

Description The `Heap check` command checks the consistency of the current application heap, which is by default the heap referenced by the global variable `theZone`.

`check` Check the consistency of the current application heap. This manual has divided the `Heap` command into three parts—display, check, and summarize—for clarity of presentation, though in fact it is one command with three options. `Display` is the default if you don't specify any options, so you must specify `check` if you want to check the heap rather than display information about it.

addr The address of the heap zone header. (You can't check part of a heap.) If you omit this parameter, Heap displays the heap pointed to by the global variable `theZone`.

`Heap check` performs range checking to make sure all pointers are even and non-NIL, and that block sizes are within the range of the heap. It verifies that the self-relative handle points to a master pointer referring to the same block. For nonrelocatable blocks, it checks if the heap zone pointer points to the zone where the block exists. `Heap check` also verifies that the total amount of free space is equal to the amount specified in the header, and that all pointers in the free master pointer list are in the heap.

Examples

This example intersperses `Printf` commands with `Heap check` commands to show at which memory locations the heap is okay.

```
Printf "Checking %P's heap at $%.08X\n", ^Pstring( $910)^, TheZone
Heap check TheZone

Printf "Checking the System heap at $%.08x\n", SysZone
Heap check SysZone

Printf "Checking the MultiFinder heap at $%.8x\n", **$2a6+$c
Heap check $2a6^^+$c

#-- output

Checking SADE's heap at $00146EF2
  The heap is okay.
Checking the System heap at $00001400
  The heap is okay.
Checking the MultiFinder heap at $00023d64
  The heap is okay.
```

See also `Heap`, `Heap totals`

Heap totals—displays heap summary

Syntax `Heap totals` [*addr*] [*blocktype*]

Type Command

Description The `Heap totals` command summarizes the state of the current application heap, which is by default the heap referenced by the global variable `theZone`. You can specify another heap that starts at *addr*.

Information is given for free, nonrelocatable, and relocatable objects. If you wish to restrict the display to a particular type of block, use the *blocktype* parameter.

`totals`	Summarize the state of the current application heap. This manual has divided the `Heap` command into three parts—display, check, and summarize—for clarity of presentation, though in fact it is one command with three options. Display is the default if you don't specify any options, so you must specify the `totals` option if you want to summarize the state of the heap rather than display information about it.
addr	The beginning address of a heap. If you omit this parameter, `Heap` displays the heap pointed to by the global variable `theZone`.
blocktype	You can specify one of the following *blocktypes* to limit the display to a particular type of block:
`purge[able]`	Limits the display to purgeable blocks.
`nonreloc[atable]`	Limits the display to nonrelocatable blocks.
`reloc[atable]`	Limits the display to relocatable blocks.
`free`	Limits the display to free blocks.
`lock[ed]`	Limits the display to locked blocks.
res[ource]	Limits the display to resources.
`restype` '*type*'	Limits the display to the specified resource type.

- *Note:* The resource type is case-sensitive and you must enclose it in single quotation marks (`'MENU'`, for example).

Example

```
Heap totals

#-- output
                          Total Blks  Total Size
Free                              23       49080
Nonrelocatable                     7        1348
Relocatable                       89       21232
  Locked & NonPurgeable            2        5796
  Locked & Purgeable               2        8136
  UnLocked & Purgeable             6         680
  UnLocked & NonPurgeable         79        6620
Heap (total)                     119       71660
```

See also `Heap, Heap check`

Help—displays help information

Type Command

Syntax `Help` [*identifier*,...]

Description The `Help` command displays information about using SADE, including the syntax of all SADE commands. To see what values *identifier* can have, just enter `Help`.

If...End—conditionally executes commands

Syntax

```
If Boolean [then]
   commands
[elseif Boolean [then]
   commands ] ...
[else
   commands ]
End
```

Description

The `If...End` construct allows for conditional execution of sequences of SADE commands. Each `If` must be concluded by a corresponding `End`. Although `elseif` and `else` are optional, they must appear between the `If` and `End` in the order indicated in the syntax description. You can use more than one `elseif` but at most one `else`.

The commands controlled by an `If` extend to the corresponding `End`, or to the first corresponding `elseif` or `else`. The commands controlled by an `elseif` extend to the next corresponding `elseif`, `else`, or `end`. The commands controlled by an `else` extend to the corresponding `End`.

When an `If...End` construct is evaluated, if the `If` *Boolean* is true, the statements controlled by the `If` are executed and the remainder of the construct to the `End` keyword is skipped. If the *Boolean* is false, the statements controlled by the `If` are skipped and the next (`elseif`) condition is checked, if present. If an `elseif` condition is evaluated and is true, the commands it controls are executed and the remainder of the construct is skipped. If no conditions are evaluated as true, when the `else` command is reached (if present), the commands controlled by the `else` are executed (otherwise, they're skipped).

You can nest `If...End` constructs.

`If` *Boolean* — Determines if the specified commands are to be executed. If the expression specified by *Boolean* evaluates to true, the commands are executed; otherwise SADE skips them and checks for the next `elseif`, `else`, or `End`, in that order.

`elseif` *Boolean*	Determines if the specified commands are to be executed. If the `If` condition is evaluated as false, SADE checks the first `elseif` condition. If the expression specified by *Boolean* evaluates to true, the commands are executed; otherwise SADE skips them and checks for the next `elseif`, `else`, or `End`, in that order.
`else`	Determines commands to be executed if the `If` condition and all `elseif` conditions evaluate to false.
`then`	Precedes the commands to be executed if the specified condition evaluates to true. It is optional and is provided for consistency with Pascal usage.
`End`	Specifies that there are no more commands to be executed nor conditional expressions to be evaluated. `End` is required.
commands	One or more SADE commands to be executed.

Example

This example steps through a five element array using a low to high index, looking for the first true element, and resetting it to false. The first four elements are true and the last one is false.

```
Define global Test[5] := (4 of 1,0)

Proc IfDemo
  Define WhichTest
  Define ShowMe
  For WhichTest:= 1 to 5 do
    Printf "\n"
    For ShowMe:= 1 to 5 do
      Printf "%t", test[showMe]
    End
    Printf " - "
    if Test[1]
      "Test[1] true"
    elseif Test[2]
     "Test[2] true"
    elseif Test[3]
     "Test[3] true"
    elseif Test[4]
     "Test[4] true"
    elseif Test[5]
```

```
     "Test[5] true"
    else
     "all tests false"
    End
    Test[ShowMe] := 0
  End
End

IfDemo

#-- output

11110 - test[1] true

01110 - test[2] true

00110 - test[3] true

00010 - test[4] true

00000 - all tests false
```

This example shows a SADE procedure that steps until the menuItem variable is false.

```
Proc StepWhile
      If menuItem =  1
            Step onEntry StepWhile
      Else
            Stop
      End
End
Step onEntry StepWhile
```

Kill—terminates an application

Syntax `Kill` [*filename*]

Type Command

Description If you want to terminate an application, (whether it's suspended or not), use the `Kill` command. Be aware, though, that `Kill` is dangerous, since it doesn't give the unlucky application a chance to perform its usual exit routines (like saving data). `Kill` does perform cleanup activities like freeing the MultiFinder memory occupied by the application and removing its trap patches.

filename The name of the application or tool to terminate. It is a string expression and you must enclose it in quotation marks if it's a string constant. The filename parameter is optional; if you omit it, SADE simply terminates the current target application.

If you are debugging an MPW Tool, the MPW Shell, not the tool itself, is SADE's target. Therefore, if you want to kill a tool that is targeted but leave MPW running, you cannot use the `Kill` command. For this purpose, SADE provides a procedure named `KillTool`;that suspends the execution of a tool but leaves MPW running. The `KillTool` procedure is defined in the `SADEStartup` script, so it is always available. To use the procedure, simply execute the command `KillTool` from the Worksheet window. You can also kill a tool but leave MPW running by choosing the Kill menu command from the File menu. If a tool is being debugged when the Kill menu item is chosen, SADE calls `KillTool` .

Examples

The command in the first example terminates execution of the target program Sample. The second command defines a variable, ExPath that specifies the path to the Sample application. `Kill` terminates Sample by using `Concat` to concatenate the directory path with the application name.

```
Kill
```

```
Define ExPath := "Volume1:CExamples:"
Kill Concat(ExPath, "Sample")
```

Leave—exits from a looping construct

Syntax `Leave [if` *Boolean*`]`

Description The `Leave` command lets you exit from a `Loop`, `While`, `Repeat`, or `For` construct. You can specify a condition for leaving with the `if` parameter.

`if` *Boolean*	Determines whether or not you exit the loop. If the expression specified by *Boolean* evaluates to true, then you exit the loop, otherwise continue to repeat the commands in the loop.

Example

In this example `testNum` is set to 0 and the loop will increment it by one until it reaches 6. However, the `Leave` command sets a condition to exit if `testNum` equals 3, so the loop terminates after `testNum` reaches 3.

```
Define testNum := 0
Repeat
  Printf "testNum = %d\n", TestNum
  Leave if testNum = 3
  testNum := testNum + 1
Until testNum = 6
Printf "after leaving Repeat loop - testNum = %d\n", testNum

#-- output

testNum = 0
testNum = 1
testNum = 2
testNum = 3
after leaving Repeat loop - testNum = 3
```

See also `Cycle`, `For`, `Loop`, `Repeat`, `While`

Length—returns length of a string

Syntax `Length ( string )`

Type Built-in function

Description `Length` returns the length in bytes of the specified string.

See also `SizeOf`

List—lists processes, tracepoints, and breakpoints

Syntax

```
List process
or
List trace [traps | addrs]
or
List break [traps | addrs]
```

Type Command

Description The `List` command shows the active processes, breakpoints, and tracepoints.

`process`	Show the active processes.
`trace`	Show the tracepoints that have been set.
`break`	Show the breakpoints that have been set.
`traps`	Limit the display to trap breakpoints (or tracepoints) only.
`addrs`	Limit the display to address breakpoints (or tracepoints) only.

For processes, the display includes the following information: a process number, a creator, and the filename for the process.

For breakpoints and tracepoints, `List` displays the location along with its symbolic representation.

Examples

The commands in this example list the breakpoints that have been set and the the processes that are active.

```
List break          # list all breakpoints set

#-- output

  EVENTLOOP.(5)     <has break action>
  EVENTLOOP.(11)
  EVENTLOOP.(2)

List Process
```

```
#-- output

Process#, creator, FileName
       2, 'MACS', 'Finder'
       3, 'MSWD', 'Microsoft Word'
       4, 'sade', 'Sade'
       5, 'MPNT', 'MacPaint'
     s 7, '????', 'Sample1'
```

See also `Trace, Break`

Loop...End—repeats commands until Leave

Syntax

```
Loop
   commands
End
```

Type Command

Description The `Loop...End` construct provides unconditional looping. The enclosed commands are executed repeatedly. To exit the loop, use the `Leave` command.

You can nest `Loop` constructs.

▲ **Warning** If you are a new SADE user, you may be tempted to try to step through a program using a simple `Loop...End` structure. This will not work. To step through a program using a script or a block of SADE commands, you must use the `onEntry` command, described later in this chapter. The following procedure uses `onEntry` to step a given number of times (in this case ten times); the `onEntry` command tells SADE what to do after each step. In this case, `onEntry` instructs SADE to take another step.

```
define SADELooper := 10
Proc StepN
      If SADELooper > 0
            SADELooper := SADELooper - 1
            where (Δpc)
            Step onEntry StepN
      Else
            Stop
      End
End
StepN
undefine SADELooper
```

To run `StepN` again, remember to reset the `SADELooper` variable to the number of steps you want. ▲

Example

```
Define Inner := 0
Define Outer := 0
Loop
  Loop
    Inner := Inner + 1
    Printf "Inner: %d " , Inner
    Leave if Inner > Outer
  End
  Inner := 0
  Outer := Outer + 1
  printf "Outer: %d\n", Outer
  leave if Outer > 4
End
```

```
#-- output
```

```
Inner: 1 Outer: 1
Inner: 1 Inner: 2 Outer: 2
Inner: 1 Inner: 2 Inner: 3 Outer: 3
Inner: 1 Inner: 2 Inner: 3 Inner: 4 Outer: 4
Inner: 1 Inner: 2 Inner: 3 Inner: 4 Inner: 5 Outer: 5
```

See also Leave

Macro—defines a macro

Syntax `Macro` *name string-expr*

Type Command

Description The `Macro` command associates a string of characters with a name. Macros allow you to define short, familiar names to use instead of long, unfamiliar strings. For instance, the SADEStartup file defines macros that let you use MacsBug-like syntax for certain SADE commands.

name — The name to assign to the macro.

string-expr — The string of characters to associate with the macro. You must enclose it in quotation marks. If you specify a string constant (such as a directory name) you must enclose it in two sets of quotation marks.

Macro definitions can be nested; that is, they can contain references to other macros. Macro definitions cannot be recursive, however; in other words, a macro definition can't reference itself. Macro definitions are not allowed in structured statements. Macros may be redefined. Macro definitions are limited to a length of 254 characters.

Examples

```
Macro br 'break'
Macro clr  'Unbreak all'

Macro dir "'volume:very:long:directory:path'"
directory dir
```

MoveWindow—moves window to location

Syntax `MoveWindow [to` *h, v*`][`*window*`]`

Type Command

Description Moves the upper-left corner of the specified *window* to the specified location.

The coordinates (0,0) are located at the left side of the screen at the bottom of the menu bar. If the location specified would place the window's title bar entirely off the visible screen, an error is returned. If you do not specify a window SADE assumes the target window (the second window from the front). If you do not specify a location, SADE returns the window's location without any effect on the window.

window	The name of the window to move. If you do not specify a window, SADE uses the target window (second from the front).
`To` *h, v*	The keyword `To` followed by the horizontal and vertical coordinates (in pixels) of the new location in that order. They are nonnegative integers. Separate them by a comma.

Examples

This example moves the target window's upper-left corner to a point approximately one inch in from the upper-left corner of the screen, and one inch below the bottom of the menu bar. (There are about 72 pixels per inch on the Macintosh display screen.)

```
MoveWindow to 72, 72
```

The next example returns the location of the target window, which would be 72, 72 when executed after the command in the first example.

```
MoveWindow
# output-
# MOVEWINDOW 72 72
```

See also `SizeWindow, WindowSize`

NaN—not a number

Syntax `NaN` (*expression*)

Type Built-in function

Description `NaN` converts the specified expression into a SANE 10-byte extended value.

OnEntry—sets commands for SADE entry

Syntax `OnEntry` [*exception-action*]

Type Command

Description The `OnEntry` command supplies commands that are to be executed each time SADE is entered. Each `OnEntry` command replaces the commands specified by the previous `OnEntry` command. In the SADEStartup file, the `StandardEntry` procedure is specified as an `OnEntry` exception action. You can define your own `OnEntry` actions in the SADEUserStartup file, but you'll probably want to use the one in SADEStartup as a model (so as not to lose the operations it performs).

exception-action	The commands to be executed each time SADE is entered. Because the `OnEntry` command accepts only one command or procedure invocation, if you want to execute multiple commands and/or procedures upon entry, use the `Begin..End` construct to group them.

See also `Break`, `Begin`

Open—opens a file

Syntax `Open [source] [behind]` *filename*

Type Command

Description The `Open` command opens the specified file.

`source`	Open the file as read-only. If this option is omitted, the window is opened as a read-write window.
`behind`	Open the window behind the front most SADE window, otherwise, it's opened as the front most window.
filename	The name of the file to open. If it is in the current directory, you can specify the name only; otherwise, specify a complete or partial directory name as well as the filename. The file must be of type 'TEXT' or SADE will be unable to open it. *Filename* is a string expression and must be enclosed in quotation marks if it's a string constant.

Example

```
Open source 'myFile'
```

See also `Close`, `Save`

Printf—prints formatted output

Syntax

`Printf` [*format*[, *argument*]...]
or
`Printf` [(*format*[, *argument*]...)]

Type

Command

Description

The `Printf` command prints string characters to a SADE window or file. You can also use `Printf` to convert values—for example, convert decimal characters to hexadecimal—and to specify how the output appears.

format — A string containing characters to print, as well as format specifications for arguments that follow. Format specifications are preceded by the % character. For instance, to print the value of the variable `myVar` as a decimal number along with a message (assuming its value is 5), enter:

```
Printf "The value of myVar is %d", myVar
# output—
The value of myVar is 5
```

Remember that the format specification is a string so you must enclose it, including the percent sign, in quotation marks.

argument — The value to be printed; it can be an actual numeric value or a variable whose value you want to print.

Each format specification applies to zero or more arguments. When the format specifications are exhausted, any remaining arguments are ignored. Likewise, when the specified arguments are exhausted, any remaining format specifications are ignored.

The following fields are used in the format specification:

[*chars*] `%` [*flags*] [*width*] [*precision*] *op*

chars — An optional sequence of characters to be printed out as is that allows you to specify a message to go with the value you are printing.

flags An optional sequence of characters which modify the meaning of the main (*op*) conversion specification:

- Left-justify within the field width rather than right-justify if the converted value has fewer characters than the specified minimum field width.

+ Always generate a "+" or "-" sign when converting signed arguments. Note, that negative values are always preceded by a "-" regardless of whether the "+" flag is specified.

space Generate a space for positive values and "-" for negative values. This space is independent of any padding used to left or right-justify the value. The "+" flag has precedence over the space flag.

Modify the main conversion operation. The modifications performed are described in conjunction with the relevant main conversion operations discussed later.

width An optional *minimum* field width, specified as a decimal integer constant (that doesn't begin with a 0) or an *. In the latter case a corresponding argument specifies the minimum field width. If the converted value has fewer characters than the width, it is padded to the width on the left (default) or right (if the - flag is specified) with spaces (default). If the converted value has more characters than the width, the width is increased to accommodate it. For %t conversions, the width specifies the minimum width to reserve for RECORD type field names.

precision The optional precision is specified as a . followed by an *optional* decimal integer or as an *. In the latter case a corresponding argument specifies the repetition count. If the decimal integer or * following the . is omitted, the precision is assumed to be 0. Precision is used to control the number of digits to be output for numeric conversions or characters for string conversions. Omitting the precision has a default value which is a function of the main conversion to be performed.

op The required main conversion operation. Table II-1 lists the single character conversion codes and the conversion that each performs. Following the table is a more detailed description of each conversion code, including how the precision and flags work for that particular conversion.

■ Table II-1 Printf operation codes

Operation Code	Conversion result
`d`	Convert to a *signed* decimal value (floating point values are truncated).
`u`	Convert to an *unsigned* decimal value (floating point values are truncated).
`X,x`	Convert to an *unsigned hexadecimal* value. A lowercase `x` causes lowercase letters (`abcdef`) in the conversion; likewise an uppercase `X` means uppercase letters (`ABCDEF`) in the conversion.
`b`	Convert to an *unsigned binary* value.
`o`	Convert to an *unsigned octal* value.
`f`	Convert to a signed decimal *floating point* value of the form [-]*ddd*`.` *ddd*, [-]`INF`, or [-]`NAN`(*ddd*) (where *ddd* is the NAN code) depending on the value.
`E,e`	Convert to a signed decimal *floating point* value of the form [-]*d*`.` *ddd*`e`±*dd* (for `e` conversion), [-]*d*`.` *ddd*`E`±*dd* (for E conversion), [-]`INF`, or [-]`NAN`(*ddd*) (where *ddd* is the NAN code) depending on the value.
`G`	Convert to a signed decimal *floating point* value. The value is converted using `f` or `e` conversion (or in the style `f` or `E` conversion when `G` is specified). The form of conversion depends on the value being converted.
`c`	Convert to a character (the value mod 256 is used).
`s`	Copy to the output as is.
`P`	Copy to the output as is; the corresponding argument must be a Pascal string or a pointer.
`t`	Convert the corresponding argument as a function of its type.
`%`	Output a single %; no parameter is used.

The rest of this section shows more information about the op codes, such as how the flag options and precision work with each one.

`d` Convert the corresponding parameter to a *signed* decimal value (floating point values are truncated).

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	explicit + or -
space	space for positive value
#	ignored

`u` Convert the corresponding parameter to an *unsigned* decimal value (floating point values are truncated).

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	ignored
space	ignored
#	ignored

`X,x` Convert the corresponding argument to an *unsigned hexadecimal* value. The argument's type determines the number of bytes that are converted. If you specify a lowercase `x`, `Printf` uses lowercase letters (`abcdef`) in the conversion; likewise, if you specify an uppercase `X`, `Printf` uses uppercase letters (`ABCDEF`) in the conversion.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	ignored
space	ignored
#	prefix converted value with a $

`b` Convert the corresponding argument to an *unsigned binary* value. The argument's type determines the number of bytes that are converted.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	ignored
space	ignored
#	ignored

`o` Convert the corresponding argument to an *unsigned octal* value. The argument's type determines the number of bytes that are converted.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	ignored
space	ignored
#	prefix converted value with a 0

`f` Convert the corresponding argument to a signed decimal *floating point* value. `Printf` converts the value to the form [-]*ddd*`.` *ddd*, [-]`INF`, or [-]`NAN`(*ddd*) (where *ddd* is the NAN code) depending on the value.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the # flag). The default precision is 6.

flags

-	left-justify
+	explicit + or -
space	space for positive value
#	force decimal point in the case where no digits follow it

`E,e` Convert the corresponding argument to a signed decimal *floating point* value. The value is converted to the form [-]*d*`.` *ddd*`e`±*dd* (for `e` conversion), [-]*d*`.` *ddd*`E`±*dd* (for E conversion), [-]`INF`, or [-]`NAN`(*ddd*) (where *ddd* is the NAN code) depending on the value. The exponent always contains at least two digits.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the # flag). The default precision is 6.

flags

-	left-justify
+	explicit + or -
space	space for positive value
#	force decimal point in the case where no digits follow it

`G` Convert the corresponding argument to a signed decimal *floating point* value. The value is converted using `f` *or* `e` conversion (or in the style `f` or `E` conversion when `G` is specified). The form of conversion depends on the value being converted; `e` or `E` conversion is performed only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result (which can be overridden with the # flag). A decimal point appears only if it is followed by a digit (which can be overridden with the # flag)

precision The precision specifies the *total* number of significant digits. If the precision is less than 1, then 1 is assumed. The default precision is 6.

flags

-	left-justify
+	explicit + or -
space	space for positive value
#	force decimal point in the case where no digits follow it and keep trailing zeros

`c` Convert the corresponding argument to a character (the value mod 256 is used).

precision ignored

flags

-	ignored
+	ignored
space	ignored
#	ignored

`s` Unless the "#" flag is used, the corresponding argument must be a string type (or a pointer) and the value is copied to the output as is. C strings and as is (Pascal packed array of char) strings are copied until a null is encountered (for C strings) or the number of characters specified at the precision is reached. Pascal strings may be processed if the type of the argument is a Pascal string. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case a C and as is strings are output up to but not including a terminating null character and entire Pascal strings are output.

flags

-	left-justify
+	ignored
space	ignored
#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

`P` Unless the "#" flag is used, the corresponding argument must be a Pascal string type (or a pointer) and the value is copied to the output as is. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

You must use an upper-case `%P` as shown to output a Pascal string type. If you use a lower-case `%p` argument, the value displayed is output as a pointer type, which is a hexadecimal number optionally preceded by `0X`.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case the entire Pascal string is output.

flags

-	left-justify
+	ignored
space	ignored
#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

`t` Convert the corresponding argument as a function of its type as follows:

a base type	`u`, `d`, `g`, `p`, or `s` as appropriate to the type with the precision and flags interpreted as a function of these format codes.
non-base type	The value(s) are displayed using a pseudo-Pascal type specification format appropriate to the type of the parameter (for example, a RECORD/struct type is displayed using a Pascal-like RECORD notation). The flags control some of the aspects of the formatted output.

The corresponding argument need not specify a value and instead may specify only a type. In this case, the type definition is displayed, again using the same pseudo-Pascal type specification format.

flags	-	display only the type even if the corresponding parameter specifies a value. The type is to be displayed exhaustively, in other words,display every type down to its base type.
	+	display only the type even if the corresponding parameter specifies a value.
	space	ignored
	#	show all values and offsets in hexadecimal

`%` Output a single %; no parameter is used.

precision	ignored	
flags	-	left-justify
	+	ignored
	space	ignored
	#	ignored

Examples

```
#-- Displays record definition of system type EventRecord.
#-- Notice how EventRecord.Where, which is of type Point, has been
#-- expanded to show its definition too.
```

```
Printf "%P", ^pstring($910)^
myProg

Printf "%P", *(pstring*) $910
myProg

Printf "%-t", EventRecord
RECORD
  WHAT: Word;
  MESSAGE: Long;
  WHEN: Long;
  WHERE: RECORD
           CASE Word OF
             (1):
               (RECORD
                  V: Word;
                  H: Word;
                END);

             (2):
               (RECORD
                  VH: ARRAY [(0, 1)] OF
                        Word;
                END);
         END;
  MODIFIERS: Word;
END


#-- Doing the same thing with a target program variable
#-- of type EventRecord. The %t format specifier lets
#-- the Printf display format be controlled by the
#-- type of the variable displayed.

typeof(myEvent)
   EVENTRECORD

Printf "%t", myEvent
RECORD
  WHAT: 3;
  MESSAGE: 16686;
  WHEN: 284053;
  WHERE: RECORD
           CASE Word OF
             (1):
               (RECORD
                  V: 272;
                  H: 267;
                END)
             (- - -);
         END;
  MODIFIERS: 2432;
END
```

Proc...End—defines a SADE procedure

Syntax

```
Proc name[ argName,... ]
  commands
End
  or
Proc name [( argName,... )]
  commands
End
```

Type

Command

Description

SADE procedures are delimited by the `Proc...End` construct. The procedure name is followed by an optional parameter list; if present, the list identifies parameters by name only. Parameters are not assigned a type but instead take on the types of the actual parameter values when the procedure is called.

name	The name of the procedure.
argName	An optional list of parameter names separated by commas. You can enclose the parameter list in parentheses if you wish, but if you do so, you must also use parentheses when calling the procedure (and vice versa).
commands	The SADE commands and language elements that compose the procedure.

When you call a procedure, the number of actual parameters that you pass need not match the number of formal parameters in the procedure definition. If you pass fewer parameters than the number of formal parameters, SADE assigns a special undefined value to the formal parameters for which there are no corresponding actual parameters.

If you pass more actual parameters than there are formal parameters, SADE ignores the extra parameters unless you use the predefined SADE variables `Arg` and `Nargs` in your procedure definition. `Arg` is an array variable that has the form `Arg[`*n*`]` where *n* is the nth actual parameter passed when the procedure is called. `Nargs` contains the number of actual parameters. These two predefined variables enable you to write flexible procedures when you don't know how many parameters will actually be passed when the procedure is called. Two of the examples show a simple for loop that uses `Arg` and `Nargs` to set multiple break points. Note that the values of these variables represent the parameter state of the currently active procedure and are not defined outside of it.

You must define a procedure before making a call to it. Therefore, if you want mutually recursive procedures, you must first define a "dummy" procedure (similar to a Pascal `FORWARD` definition). You can then write a second procedure that calls the dummy procedure, and then redefine the dummy procedure to do what you want (including calling the second procedure). The minimal dummy procedure definition is: `Proc foo; End;`.

You can nest procedure calls.

Examples

This example illustrates the use of the `Nargs` and `Arg` variables to write a procedure that will accept an indefinite number of parameters. When you call this procedure, it sets a breakpoint on each parameter that you pass. `Nargs` contains the number of the last parameter passed; `Arg` is used like an array variable to accesses each parameter by number. Therefore, the loop is in effect for the first through the last parameter passed, and each parameter is passed in turn to the `Break` command by way of the `count` and `arg` variables.

```
Proc SetBreaks ()
      Define count
      For count := 1 to Nargs
            Break Arg[count]
      End
End

# Call SetBreaks to set breakpoints on DrawWindow, EventLoop, SetLight

SetBreaks (DrawWindow, EventLoop, SetLight)
```

See also `Func`

Quit—quits SADE

Syntax `Quit`

Type Command

Description The `Quit` command terminates SADE and passes control to another process as determined by MultiFinder. Be aware that `Quit` kills any suspended applications.

See also `Shutdown`

Redirect—redirects output

Syntax

```
Redirect [append] filename
  or
Redirect [pop][all]
```

Type Command

Description The `Redirect` command redirects the output from SADE commands to the specified file.

filename	The name of the file to which output is redirected.
`append`	This parameter instructs SADE to append output to the end of the specified file rather than replacing the contents of the file. You can also use the § character (Option-6) to replace or append to the selection in an open window (see the example below).
`pop`	If you have nested `Redirect` commands to create a queue of files, this parameter redirects the output of SADE commands to the file at the head of the queue (the last file specified).
`all`	This parameter (by itself or used with `pop`) redirects output to the current command window.

▲ **Warning** Be aware that if you use use `Redirect` to replace the contents of a file that's not open, there's no way to undo it. (If the file is open, you can close it and respond "No" when the dialog asks whether to save changes.) ▲

You can nest `Redirect` commands to as many as 10 different files; SADE maintains the names of these files as a last-in, first-out queue. If you use the `pop` keyword, or if you use no parameters at all, the output from SADE commands is redirected to the file at the head of the queue. If you specify `all` or `pop all`, standard output is redirected to the current command window.

- *Note:* Any error conditions cause SADE to perform an implicit `pop all` for any redirected files; this ensures that output returns to the current command window.

Examples

In this example, `Redirect` replaces the current selection.

```
Redirect "whyNot.$"   # Replace the current selection
```

In this example, the `Redirect` command creates a file to hold output from SADE. Entering a string echoes the string. That output is redirected to the file, becoming the file's contents. Here the string is a comment and a SADE command to execute the contents of the next file in the chain.

```
Open 'exec1'
Redirect 'exec1'

'"\n executing exec1 now"'
"Execute 'exec2'"

Open "exec2"
Redirect 'exec2'
"'Now executing exec2'"
"Execute 'exec3'"

Open "exec3"
Redirect 'exec3'
'"Done in exec3"'

Redirect pop all
Execute "exec1"

Alert "Try a tile windows here\n∂
Then look at the worksheet for output"

#-- output

Executing exec1 now
Now executing exec2
Done in exec3
```

Repeat...Until—conditionally repeats commands

Syntax

```
Repeat
   commands
Until Boolean
```

Type Command

Description The `Repeat ... Until` construct provides conditional looping with a test at the end of the loop. The enclosed commands are executed until *Boolean* is true. The enclosed commands are executed at least once.

`Repeat` constructs may be nested.

commands	The commands to be executed.
`until` *Boolean*	Determines the condition on which execution of the enclosed commands is terminated. The enclosed commands are executed until *Boolean* is true.

Example

```
Define count := 0
Repeat
   Printf "%d\n", count
   count := count + 1
Until count = 5

#-- output

0
1
2
3
4
```

See also `Leave`

Request—displays request dialog box

Syntax `Request ( string[, string] )`

Type Built-in function

Description `Request` returns a string after displaying a request dialog box. The dialog presents an OK button and a Cancel button. The first string argument appears in the dialog box as the request message. The second, optional string argument specifies a default string to present in the request box.

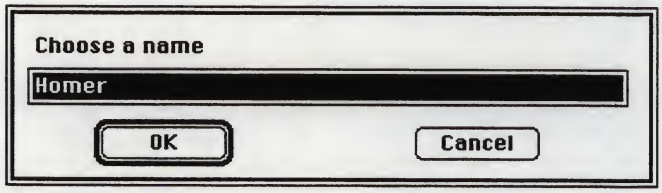
When you click Cancel, `Request` returns the string "_CANCEL_". Otherwise, it returns the string that you enter in the box.

Example

This example creates the dialog box shown in Figure II-2

```
Request('Choose a name','Homer')
```

■ **Figure II-2** Request dialog box



See also `Confirm`

Resource—displays the resource map

Syntax `Resource [display] [addr] [restype 'type']`

Type Command

Description The `Resource` command displays the contents of your application's resource maps and the system resource map. If you want to display only a particular map, *addr* should be the address of the map. The information displayed for each map includes: its location, the resource ID, the resource type, the value of the master pointer, whether the resource is locked or unlocked, and the resource name. If a resource isn't loaded, the master pointer field shows "NotLoaded."

You can also restrict the display to a particular resource type by using the `restype` keyword with the desired *type*. Note that *type* is case-sensitive and should be enclosed in single quotation marks (`'WIND'`, for example).

`display`	Display the application's resource map and the system resource map. The `Resource` command actually has two parts—display and check—determined by the `display` and `check` options, but the manual has divided `Resource` into two separate commands (for ease of presentation). The default, if you don't specify any options, is to display the resource maps, so you can omit this option if you wish.
addr	The address of a particular map. If you omit *addr*, SADE displays the map for all resources
`restype 'type'`	Restrict the display to a particular resource type. *Type* is a standard Macintosh resource type.

Example

```
Resource restype 'MENU'

#-- output

Resource Map at $00501E8C
   ResId  RType  MasterPtr  Locked?   Name
     128  MENU   $00501E8C  Unlocked
     129  MENU   $00501EC0  Unlocked
     130  MENU   $00501F50  Unlocked
     131  MENU   $00501FA0  Unlocked
Resource Map at $0010CF68
   ResId  RType  MasterPtr  Locked?   Name
   33024  MENU   NotLoaded
   49046  MENU   $0010EC94  Unlocked
   49047  MENU   $001683F8  Unlocked
   49063  MENU   $000DF76C  Unlocked
   49064  MENU   $0010E764  Unlocked
   49065  MENU   $0010E794  Unlocked
   61536  MENU   NotLoaded
```

See also `Resource check`

Resource check—checks the resource map

Syntax `Resource check` [*addr*]

Type Command

Description The `Resource check` command checks the target application's resource maps for consistency. If you want to check a particular map, *addr* should point to the address of the map. If an inconsistency is found, the command displays a diagnostic message specifying the problem.

`check`	Check the application's resource map for consistency. The `Resource` command actually has two parts—display and check—determined by the `display` and `check` options, but the manual has divided `Resource` into two separate commands (for ease of presentation). The default, if you don't specify any options, is to display the resource maps, so you must specify the `check` option if you want to check the resource map rather than display it.
addr	The address of a particular map. If you omit *addr*, SADE displays the map for all resources.

See also `Resource`

Return—returns from a procedure or function

Syntax `Return [ result ]`

Type Command

Description The `Return` command returns you from a procedure or function currently in execution. When returning from a function, the function *result* must be specified. (When returning from a procedure, there is no return value.)

result	An expression that specifies the return value.

Example

```
Define global SadeArray[4] := (1, "this is two", 3.3, 4)
Func MiscTypes (index)
    Return SadeArray[index]
End

Define selector
For selector := 1 to 4 do
    Printf "%t \n", MiscTypes(selector)
End

#-- output

1
this is two
3.3
4
```

See also `Func`, `Proc`

SADEKey—defines a key for entering SADE

Syntax `Sadekey [ keycode ]`

Type Command

Description The SADEKey command lets you specify a different Command-Option key combination for entering SADE. To see what key is specified as the SADEKey, just type `SADEKey`.

keycode	A keycode for the SADEKey. A complete list of keycodes can be found in the Toolbox Event Manager chapter of *Inside Macintosh, Volume V.*

Example

```
sadekey $33     #-- define Command-Option-Delete combination as SADEKey
```

Save—saves a file

Syntax `Save [all | filename]`

Type Command

Description The `Save` command saves the specified file or, if `all` is specified, saves all files. If the specified file wasn't modified since the last time it was saved, `Save` does nothing.

If no parameters are given, `Save` saves the target window.

filename	The name of the file to save. If you omit this parameter, `Save` saves the target window. *Filename* is a string expression and you must enclose it in quotation marks if it's a string constant.
`all`	Save all files.

Example

```
Save 'myFile'
```

See also `Open`, `Close`

Selection—returns text of current selection

Syntax `Selection ( windowName )`

Type Built-in function

Description `Selection` returns the text of the current selection in the specified window. The value returned is of type PString. To get the value of the string if it contains a name or expression (for instance, to set a breakpoint), apply the `Eval` function on the string.

Example

This example returns the string that is highlighted in the active window, that is, the frontmost window.

```
Selection(ActiveWindow)
```

See also `Eval`

Shutdown—shuts down or restarts the machine

Syntax `Shutdown [ restart ]`

Type Command

Description The `Shutdown` command terminates SADE and calls the Shutdown Manager. Be aware that all unsaved work is lost.

`restart`	Restart the Macintosh after shutting down.

SizeOf—returns size of variable, type, or argument

Syntax `SizeOf` (*variable* | *type* | *argument*)

Type Built-in function

Description `SizeOf` returns the number of bytes occupied by a variable or type. You can also use `SizeOf` to determine the size of an argument to a SADE procedure or function. You cannot use `SizeOf` with SADE array variables.

Example

This example shows the size of the SADE type, Long.

```
SizeOf(long)
#output-
4
```

See also `Length`

SizeWindow—sets a window's size

Syntax `SizeWindow[to` *h, v*`][`*window*`]`

Type Command

Description `SizeWindow` sets the size of the specified *window* to the specified horizontal and vertical dimensions. If the size specified would cause the window to be too big for the screen, SADE returns an error message.

window	The name of the window to size. If you do not specify a window, SADE uses the target window (second from the front).
`to` *h, v*	The keyword `To` followed by the horizontal and vertical dimensions (in pixels) of the window in that order. They are nonnegative integers. Separate them by a comma.

Examples

The command in this example makes the target window 200 pixels square in size. Note that a there are about 72 pixels per inch on the Macintosh display screen.

```
SizeWindow To 200, 200
```

The `SizeWindow` command in this example has no parameters so it displays the size of the sample.c window:

```
SizeWindow 'sample.c'
# output-
# SIZEWINDOW 630 393 sample.c
```

This command makes the Worksheet window 500 x 100 pixels in size. Note the use of the built-in `Concat` function to concatenate the directory path defined by `sadedir` with the name of the SADE Worksheet.

```
SizeWindow To 500, 100 Concat(sadedir,'SADE Worksheet')
```

See also `MoveWindow, WindowSize`

SourcePath—tells SADE where your source files are

Syntax `Sourcepath [[ add | del[ete ]] directoryName, ... ]`

Type Command

Description The `SourcePath` command tells SADE what directory your source files are in. If your source files are in more than one directory, you can give the `SourcePath` command a list. If you're unsure which directories you've specified, simply enter `Sourcepath` and the current search path is displayed.

`add` Add a directory to the list of directories in the search path.

▲ **Warning** If you specify a directory without using the `add` keyword, any directories previously specified are replaced. ▲

`del[ete]` Delete a directory from the list of directories in the search path.

directoryName The name of the directory containing your source files. You can specify a list of directory names separated by commas if your source files are contained in more than one directory.

Examples

```
Sourcepath 'srcdir', ':otherdir'  # sources in more than one directory
Sourcepath add ":samples"         # add directory Samples to search path
```

See also `Directory`

SourceToAddr—returns address of source statement

Syntax `SourceToAddr ( windowName[, errorFlag] )`

Type Built-in function

Description `SourceToAddr` returns the address corresponding to the statement selected in the specified window. The window need not be active. `SourceToAddr` displays the address symbolically; `SourceToAddr` returns zero if it cannot determine the address.

If you pass a nonzero value in the optional *errorFlag*, `SourceToAddr` returns a string describing why the address could not be found.

Example

In this example, `SourceToAddr` returns the address of the statement that a user has highlighted in the target (source code) window. If the user has highlighted code that does not represent a statement, that is, if `SourceToAddr` returns a Pstring, then SADE returns an error to the user. Note that this code is similar to code in the SADEStartup file that implements the Statement Selected Is? item in the SourceCmds menu.

```
foo := SourceToAddr(targetWindow,1)
If TypeOf(foo) = 'PString' then
#report error in foo
else
#foo has address
```

See also `AddrToSource`

Stack—displays stack frames

Syntax `Stack [`*count*`][ at` *addr*`]`

Type Command

Description The `Stack` command displays a list of the stack frames for the target application. The stack frames displayed are based on register A6 or the *addr* parameter if `at` is specified.

For each entry, `Stack` gives the address of the stack frame, the name of the procedure or function that allocated the frame, and the name and offset (if available) of the parent procedure.

count	The maximum number of stack frames to display counting back from the current frame).
`at` *addr*	The register on which to base the stack frame display. `Stack` uses register A6 if you omit this parameter.

Example

```
Stack at DisplayText.(6)
Stack
Frame Addr  Frame Owner   Called From
<main>      %__MAIN
$0032BC24   main          %__MAIN+$0028
$0032BB2C   SkelMain      main.(51)
$0032BB0C   LogEvent      SkelMain.(13)+$0012
$0032BADC   ReportUpdate  LogEvent.(50)+$0004
$0032BACC   DisplayText   ReportUpdate.(1)+$0004
```

Step—executes single step

Syntax `Step[asm | line][into][onEntry` *commands*`]`

Type Command

Description The `Step` command lets you execute your program one step at a time, from either the source code level or the object code level.

`asm` Perform object-code (assembly-language) debugging. The default, if you omit this parameter, is source-level debugging.

`line` Perform source-level debugging. This is the default, so you can omit this option if you wish.

`into` Step into subroutines. The default, if you omit this parameter, is to step over subroutines.

`onEntry` *commands*
Specify actions to occur upon returning to SADE after stepping the program. This option is required if you want to execute more than one step at a time as part of a script or SADE procedure. See the examples.

If you specify `line` (the default), execution proceeds one source statement at a time.

If you specify `asm`, execution proceeds one instruction at a time; the instruction at the program counter is executed and SADE is re-entered. SADE always treats traps as single instructions; SADE steps over them, stopping at the first instruction following the trap. You can either step into or over subroutines called by JSR and BSR instructions. If you specify `into`, SADE steps in, stopping at the first instruction of the subroutine. If you omit `into`, SADE treats `BSR` (branch to subroutine) and `JSR` (jump to subroutine) instructions as single instructions.

▲ **Warning** Don't try to step over a routine that does not return to the caller; for instance, a call to `longjmp()`. SADE steps over procedure and function calls by setting the trace bit until after the `JSR` or `BSR` is executed, and then replaces the return address with the address of a SADE routine. Since `longjmp()` restores a previously saved register set, including a new stack pointer, SADE's return is lost. If you want to go to the routine restored by longjump0, execute `Step asm into` until the registers have been modified and then do a source step. Or better still, if you know where the jump will go, set a breakpoint in that routine. ▲

Use `onEntry` to specify an action to occur when SADE is reentered following a step—for example, to display the value of a variable. If you do not use `onEntry` but execute the `Step` command as part of a group of commands (in a script file or highlighted in the Worksheet) SADE evaluates the other commands first before executing the `Step` command regardless of the order in which you place the commands in the script or in the Worksheet. This means, for example, if you place a command such as `Printf` after a `Step` command to display the value of a variable, and execute the commands together, `Printf` displays the value of the variable before, not after, the step.

△ **Important** You cannot execute more than one `Step` command at a time, whether as part of a script or as a highlighted group of commands unless you use the `onEntry` keyword. With `onEntry` you can tell SADE to execute another `Step` command after returning to SADE following the first step. (See the last example.) △

Example

In the first example, if the two commands are executed together, SADE displays the value of `newStopped` before stepping the program (remember that SADE evaluates all other commands in a script before executing a `Step` command).

```
Step
Printf "The value of newStopped is %d",newStopped
#newStopped before step
```

The second example shows how to use the `onEntry` keyword to display the value of a variable after stepping. Note in the second example that the `Printf` command is part of the `Step` command—it belongs to the action specified by `onEntry`—so it must be on the same line as `Step` or, as in this case, made part of the line by the continuation character (∂).

```
Step onEntry ∂
    Printf "The value of newStopped is %d",newStopped
#newStopped after step
```

You can use the `Begin..End` construct to execute multiple commands as part of the action specified by `onEntry`. This example displays the value of three different variables after stepping.

```
Step onEntry Begin
      Printf "The value of newStopped is %d\n",newStopped
      Printf "The value of gStopped is %d\n",gStopped
      Printf "The value of menuID is %d\n",menuID
End
```

Unless you use the `onEntry` keyword, a script can only contain one `Step` command (actually, only one execution command, that is, one of `Go`, `Step`, or `Target`). Using `onEntry` and the `Begin..End` construct, you can create scripts or procedures that nest `Step` commands to multiple levels, and execute other commands after each step, as in the following example.

```
Proc stepProc
"\ncurrent pc"
Disasm pc 4

"\nstep by instruction twice"
Step asm onEntry ∂
Begin
      Disasm pc 1
      Step asm onEntry ∂
      Begin
            Disasm pc 1

            "\nstep into a procedure call"
            Step into onEntry ∂
            Begin
                  Disasm pc 4

                  "\nstep by statement line twice"
                  Step line onEntry ∂
                  Begin
                        Disasm pc 4
                        Step onEntry ∂
                        Begin
                              Disasm pc 4

                              Stop
                        End
                  End
            End
      End
End
End

Target 'sample1'
Break DoMenuCommand.(15)
Go onEntry stepProc
```

Stop—terminate break action

Syntax `Stop`

Type Command

Description The `Stop` command terminates the current break action and returns you to SADE. If the current execution was within a structured statement (`Begin...End`, for instance), or if multiple commands were selected, the pending commands are executed. To terminate a break action and cancel pending commands, use the `Abort` command.

Example

```
Directory  'VolName:Path:toMyProg:'
Launch 'myProg'

Proc WhichEvent(stopType)
      Define global EventType[16] := ('null',
'     mouse-down','mouse-up',
'     key-down', 'key-up','auto-key',
'     update', 'disk-inserted','activate',
'     network', 'device driver', 'app1',' app2', 'app3', 'app4')

      If theEvent.what  = stopType then
            Printf "%P received, stopping\n", EventType[stopType+1]
            Stop
      else
            EventType[theEvent.what+1]
            Printf
      End
End

Break _waitNextEvent from applzone..applzone^ whichEvent(1)
Go

#-- output

key-down
update
key-down
mouse-down received, stopping
```

See also `Abort`, `Break`, `Quit`

Target—identify your application

Syntax `Target` [*progName*[`using` *symbolFileName*][`onEntry` *commands*]]

Type Command

Description The `Target` command tells SADE the name of the application you want to debug and identifies the symbol file (the `.SYM` file generated by the linker). If the `.SYM` file is already in the same directory as the application and is called *progName*`.SYM` (which it usually is), you don't need to bother with the `using` option. `Target` also launches the application, stopping at the first line of the `main` program, and opens the source file containing `main`.

> *Note:* If you select the Stop Before Constructor menu command, SADE stops your program as soon as it is launched instead of bringing it to the first line of `main`.

progName The name of the application to debug. Enclose it in quotation marks if it is a string constant. If you omit *progName*, SADE returns the name of the current target application.

`using` *symbolFileName*
A keyword and the name of the symbol file generated by the linker. Enclose *symbolFileName* in quotation marks if it is a string constant. If you omit this parameter, SADE assumes that the symbol file is in the same directory as the application and is called *progName*`.sym` (the default name provided by the linker)

`onEntry` *commands*
Specify actions to occur upon returning to SADE after launching the program. See the second example.

To find out what the current target is, just type `Target`.

To release the current target application (for example, if an application is targeted by SADE, you cannot relink it until you release it) use `Target` with a null string enclosed by double quotation marks, that is,

```
Target ""
```

Examples

The first example specifies the MPW Shell as the target and uses the `using` option to specify ToolStuff as the tool to debug. Note that the `Target` command automatically opens the source file and breaks at `main.(0)`.

```
Target "VolName:MPW:MPW Shell" using "VolName:MPW:ToolStuff:tool.sym"
SourcePath add "VolName:MPW:ToolStuff:"
#-- Run tool to break to SADE with tool as target
#-- Target puts pc at main.(0) automatically
```

The `Target` command in this example puts a breakpoint in the `DoMenuCommand` routine and runs the program to that point.

```
Target 'sample1' onEntry Begin
      break DoMenuCommand
      Go
End
```

Timer—returns timing values

Syntax `Timer ( [ value [, Boolean ]])`

Type Built-in function

Description `Timer` uses the global variable `TickCount` to provide timing-related functions. If you pass no arguments, `Timer` returns the current `TickCount`. If you specify a value (typically a previous value of `TickCount`), `Timer` returns the difference between that value and the current value of `TickCount` (that is, `TickCount-value`). If you also specify a nonzero Boolean value, the difference is returned as a string of the form "*sss.hh*", representing seconds and hundredths of a second. (If the Boolean is zero, it's ignored.)

Example

Execute together the lines in the following example to see how long SADE takes to launch your application.

```
Define starttime := Timer()
Target 'sample1' onentry Timer (starttime,1)
```

Trace—sets tracepoints

Syntax

`Trace` *addr*`,...`
or
`Trace` *trap*`[ from` *addrRange*`],...`
or
`Trace` *trap-range*`[ from` *addrRange*`],...`
or
`Trace all traps[ from` *addrRange*`]`

Type

Command

Description

The `Trace` command sets tracepoints on the specified address or traps within the target program. You can set tracepoints on a single trap, a range of traps, or on all traps.

addr	A RAM address or symbolic reference. If you use a symbolic reference, the code need not be in memory at the time you set the tracepoint.
trap	A trap name or number. You must prefix a trap number with the trap character (†—Option-t) and a trap name with an underscore character (`_InitGraf`, for example).
trap-range	A range of traps. Use the range operator (`..`) between trap names or numbers.
`from` *addrRange*	SADE displays the trace message only when the trap is called from the specified range.
`all traps`	Sets a tracepoint on every trap.

After setting the tracepoints, you can resume program execution. When the program encounters the tracepoint, SADE displays a message on standard output, reporting the address or trap being traced, with a symbolic representation of the address if possible. Program execution resumes after SADE displays the message.

You can specify multiple tracepoints, separated by commas, with a single `Trace` command.

To remove a tracepoint, use the `Untrace` command.

Example

These two `Trap` commands set traces on a range of traps, in the first case using trap names with the underscore character, and in the second case using the trap operator with trap numbers.

```
Trace _OpenResFile.._GetResource     #use a trap range
Trace †$A997..†$A9A0                 #use a trap range
```

See also `Break`, `List`, `Untrace`

TypeOf—return type of an expression

Syntax `TypeOf` (*expression*)

Type Built-in function

Description `TypeOf` returns a string containing the name of the type of the given expression. If SADE doesn't know the name of the type, it returns a string of the form `Type #`*n*, where *n* is SADE's internal index for the type.

Unbreak—removes breakpoints

Syntax

`Unbreak` *addr*`,...`
or
`Unbreak` *trap*`,...`
or
`Unbreak` *trap-range*`,...`
or
`Unbreak all [traps | addrs]`

Type

Command

Description

The `Unbreak` command clears the breakpoint as well as any associated break action, for the specified addresses or traps. The `all` option clears all breaks set in the target program. The `all` option can also be followed by `traps` or `addrs` to restrict the command to traps or addresses respectively.

addr	A RAM address or symbolic reference.
trap	A trap name or number. You must prefix a trap number with the trap character (†—Option-t) and a trap name with an underscore character (`_InitGraf`, for example).
trap-range	A range of traps. Use the range operator (`..`) between trap names or numbers.
`all`	Remove all breakpoints.
`all traps`	Remove all trap breakpoints.
`all addrs`	Remove all address breakpoints.

Example

```
Unbreak _GetResource    #undo break on GetResource trap
```

See also `Break`

Undef—determines if variable is undefined

Syntax `Undef` (*parameter* | *variable*)

Type Built-in function

Description `Undef` determines whether the given SADE parameter or variable has been initialized. If the parameter or variable is initialized, `Undef` returns 0; if it's not initialized, `Undef` returns 1. If you pass an undefined identifier to `Undef`, an error results.

Undefine—removes definitions

Syntax `Undefine` *name*`,...`

Type Command

Description The `Undefine` command removes the definition of the specified global SADE variable, procedure, function, or macro. You can supply a list of names to remove multiple definitions. Note that `Undefine` does not remove local variables defined within SADE procedures or functions.

name — A SADE global variable, procedure, function, or macro name.

If you want to redefine an item, you don't need to use `Undefine`; you can just assign a new value to the existing name using the `Define` command.

Example

```
Proc ControlledProc
      Printf "%d   ", ControllerGlobal
End

Proc Controller
      Define global ControllerGlobal
      Define max := 7

      For ControllerGlobal := 1 to max
            ControlledProc
      End
      Undefine ControllerGlobal
      Undefine ControlledProc
End

#-- output

Controller
2   3   4   5   6   7

ControllerGlobal
### Could not find "ControllerGlobal" as a program symbol

ControlledProc
### Could not find "ControlledProc" as a program symbol
```

See also `Proc`, `Func`, `Macro`, `Define`

Untrace—removes tracepoints

Syntax

```
Untrace addr,...
  or
Untrace trap,...
  or
Untrace trap-range,...
  or
Untrace all [traps | addrs]
```

Type Command

Description The `Untrace` command clears the tracepoint at the specified addresses or traps. The `all` keyword clears all tracepoints within the target program. The `all` keyword can optionally be followed by the `traps` or `addrs` keywords to restrict the command to traps or addresses respectively.

addr	A RAM address or symbolic reference.
trap	A trap name or number. You must prefix a trap number with the trap character (†—Option-t) and a trap name with an underscore character (`_InitGraf`, for example).
trap-range	A range of traps. Use the range operator (`..`) between trap names or numbers.
`all`	Remove all tracepoints.
`all traps`	Remove all trap tracepoints.
`all addrs`	Remove all address tracepoints.

Example

```
Untrace _GetResource    #undo trace on _GetResource trap
```

See also `Trace`

Version—displays SADE version information

Syntax `Version`

Type Command

Description The `Version` command displays the current SADE version number.

Where—returns symbolic representation of address

Syntax `Where` (*address*)

Type Built-in function

Description `Where` returns a string containing a symbolic representation of the given address.

Example

In this example, `Where` returns the source statement corresponding to the program counter (pc).

```
Where(pc)
#output-
DoMenuCommand.(31)
```

See also `AddrToSource`

While...End—conditionally repeats commands

Syntax

```
While Boolean[do]
   commands
End
```

Type

Command

Description

The `While...End` construct provides conditional looping with a test at the beginning of the loop. The enclosed commands are executed as long as *Boolean* is true. If the condition is false at the outset, the enclosed commands are never executed.

`While` constructs may be nested.

Boolean	A Boolean expression that determines whether the enclosed commands are executed. The enclosed commands are executed as long as *Boolean* is true.
`do`	Execute the commands if the Boolean expression is true. This option is not required. It is provided only for consistency with Pascal usage. As long as the Boolean expression is true, the commands will be executed whether you specify this option or not.
commands	The commands to be executed.

Example

```
Define goSmall := 10
While goSmall > -2 do
   While goSmall > 4 do
      While goSmall > 7 do
         Printf "          Inner loop - goSmall = %d\n", goSmall
         goSmall := goSmall -1
      End
      Printf "     Middle loop - goSmall = %d\n", goSmall
      goSmall := goSmall - 1
   End
   Printf "Outer loop - goSmall = %d\n", goSmall
   goSmall := goSmall - 1
End

#-- output

Inner loop - goSmall = 10
```

```
Inner loop - goSmall = 9
Inner loop - goSmall = 8
Middle loop - goSmall = 7
Middle loop - goSmall = 6
Middle loop - goSmall = 5
Outer loop - goSmall = 4
Outer loop - goSmall = 3
Outer loop - goSmall = 2
Outer loop - goSmall = 1
Outer loop - goSmall = 0
Outer loop - goSmall = -1
```

See also `Leave`

WindowSize—set size of zoom and new windows

Syntax `WindowSize zoom | new [`*top, left, bottom, right*`]`

Type Command

Description The `WindowSize` command allows you to set the size and position that all newly created windows will take, and to set the size and position a window will take when you click the Zoom box (upper right corner of the window).

`zoom | new` If you specify `new`, the coordinates that you specify determine the size and location of newly created windows in SADE.

If you specify `zoom`, the coordinates that you specify determine the size and location of windows in SADE when you press the Zoom box.

There is no default; you must specify either `zoom` or `new`.

top, left, bottom, right
The location for the four corners of the window (in pixels), in this order.

Example

This example specifies that new windows will be approximately one inch from the top and one inch from the bottom, and will be about two inches long and three inches wide. (There are approximately 72 pixels per inch on the Macintosh screen.)

```
Windowsize new 72,72,144,216
```

See also `MoveWindow`, `SizeWindow`

Index

T

U

V

W

X, Y, Z

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft Word software. Proof pages were created on Apple LaserWriter® printers. Final pages were created on the Varityper VT600 imagesetter. Line art was created using Adobe Illustrator. PostScript®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer: Michael Kline

Developmental Editor: Anne Szabla

Illustrator: Sandee Karr

Prodction Supervisor: Janet M. Anders

Special thanks to Jordan Mattson and Aliza Peleg for product management and to Tom Becker of Applied Biosystems, Inc. for creating the SADE symbol file icon and the animated bug cursor.

SADE: Symbolic Application Debugging Environment

Version 1.3

This package contains

1	Manual	*SADE Reference*
1	Set of release notes	*SADE 1.3 Release Notes*
3	Disks	*SADE,* Disk 1 of 2 *SADE,* Disk 2 of 2 *SADE System Additions*
1	1-inch binder	
13	Tab dividers	

If you have any questions, please call

800-282-2732 (U.S.)
(408) 562-3910 (International)
800-637-0029 (Canada)

M0026LL/E 7/30/91

APPLE COMPUTER, INC. SOFTWARE LICENSE

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. **License.** The application, demonstration, system and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Apple Software") and related documentation are licensed to you by Apple. You own the disk on which the Apple Software is recorded but Apple and/or Apple's Licensor(s) retain title to the Apple Software and related documentation. This License allows you to use the Apple Software on a single Apple computer and make one copy of the Apple Software in machine-readable form for backup purposes only. You must reproduce on such copy the Apple copyright notice and any other proprietary legends that were on the original copy of the Apple Software. You may also transfer all your license rights in the Apple Software, the backup copy of the Apple Software, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. **Restrictions.** The Apple Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Apple Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Apple Software in whole or in part. You may not electronically transmit the Apple Software from one computer to another or over a network.

3. **Support.** You acknowledge and agree that Apple may not offer any technical support in the use of the Software.

4. **Termination.** This License is effective until terminated. You may terminate this License at any time by destroying the Apple Software and related documentation and all copies thereof. This License will terminate immediately without notice from Apple if you fail to comply with any provision of this License. Upon termination you must destroy the Apple Software and related documentation and all copies thereof.

5. **Export Law Assurances.** You agree and certify that neither the Apple Software nor any other technical data received from Apple, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States.

6. **Government End Users.** If you are acquiring the Apple Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Apple Software is supplied to the Department of Defense (DoD), the Apple Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Apple Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and

(ii) if the Apple Software is supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Apple Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

7. **Limited Warranty on Media.** Apple warrants the disks on which the Apple Software is recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Apple's entire liability and your exclusive remedy will be replacement of the disk not meeting Apple's limited warranty and which is returned to Apple or an Apple authorized representative with a copy of the receipt. Apple will have no responsibility to replace a disk damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKS, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. **Disclaimer of Warranty on Apple Software.** You expressly acknowledge and agree that use of the Apple Software is at your sole risk. The Apple Software and related documentation are provided "AS IS" and without warranty of any kind and Apple and Apple's Licensor(s) (for the purposes of provisions 8 and 9, Apple and Apple's Licensor(s) shall be collectively referred to as "Apple") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. APPLE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE APPLE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE APPLE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE APPLE SOFTWARE WILL BE CORRECTED. FURTHERMORE, APPLE DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE APPLE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE APPLE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

9. **Limitation of Liability.** UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL APPLE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE APPLE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

In no event shall Apple's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Apple Software.

10. **Controlling Law and Severability.** This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

11. **Complete Agreement.** This License constitutes the entire agreement between the parties with respect to the use of the Apple Software and related documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Apple.

7/15/91
001-0158-A